Using Cray Performance Measurement and Analysis Tools
4.1 Published December 2007 Supports CrayPat 4.1 and Cray Apprentice2 4.1 running on Cray XT3, Cray XT4, and Cray XT5 systems, including Cray XT5_h systems with Cray X2 compute blades.

3.1 Published October 2006 First release. Supports CrayPat 3.1 and Cray Apprentice2 3.1 running on Cray XT systems.
Changes to this Document

This update of *Using Cray Performance Measurement and Analysis Tools* supports the 6.1.3 release of CrayPat, CrayPat-lite, Cray Apprentice2, Reveal and the Cray PAPI components, which collectively are referred to as the *Cray Performance Measurement and Analysis Tools*.

Added information

- This release introduces support for capturing performance data from the Intel Running Average Power Limit (RAPL) and Cray Power Management (PM) performance counters. See Power Management Counters on page 54 and the *rapl(5)* man page.
- This release introduces the *cuda, dl, sheap, and sysfs* predefined trace groups. See Using Predefined Trace Groups on page 27.
- This release introduces the new *PAT_BUILD_EMBED_RTENV* pat_build environment variable. See Advanced Users: Environment Variables and Build Directives on page 31 and pat_build Environment Variables on page 85.
- This release introduces the new *PAT_RT_ACC_ACTIVITY_BUFFER_SIZE* and *PAT_RT_RECORD* run time environment variables. See Run Time Environment Variables on page 86.
- The behavior of the *PAT_record* API call has been changed and new event categories have been added to the *PAT_counters* API call to support RAPL and PM performance counters. See API Calls on page 37.

Revised information

- The default experiment has been changed from sampling (synchronous) to Automatic Profiling Analysis. This change affects pat_build usage and also affects the behavior of the *PAT_RT_EXPERIMENT* run time environment variable. For more information, see Basic Profiling on page 25, Selecting a Predefined Experiment on page 49, and Run Time Environment Variables on page 86.

Deleted information

- The section "Using the PAPI Cray NPU Component" is removed from this guide and superseded by a separate document, *Using the PAPI Cray NPU Component* (S-0046).
- The deprecated run time environment variables *PAT_RT_HWPC, PAT_RT_NWPC,* and *PAT_RT_ACCPC,* and their related variants, are no longer supported and all references to them have been deleted.
- The **Freeze Panel** function has been removed from Cray Apprentice2.
# Contents

## Introduction [1]

1.1 Understanding Performance Analysis ........................................ 11
1.2 About CrayPat-lite ................................................................. 12
   1.2.1 CrayPat-lite Options ..................................................... 13
   1.2.2 Using CrayPat-lite ....................................................... 14
   1.2.3 Switching from CrayPat-lite to CrayPat ............................ 15
   1.2.4 Determining Whether a Binary is Already Instrumented .......... 15
1.3 About CrayPat ................................................................. 15
   1.3.1 Instrumenting the Program ............................................ 16
      1.3.1.1 Automatic Profiling Analysis .................................. 16
      1.3.1.2 MPI Automatic Rank Order Analysis ......................... 17
   1.3.2 Running the Program and Collecting Data ......................... 17
   1.3.3 Analyzing the Results .................................................. 17
      1.3.3.1 Initial Analysis: Using `pat_report` ....................... 18
1.4 In-depth Analysis: Using Cray Apprentice2 ................................ 18
   1.4.1 On Linux Systems ....................................................... 19
   1.4.2 On Microsoft Windows 7 Systems .................................... 20
   1.4.3 On Apple Macintosh Systems ......................................... 20
1.5 Source Code Analysis: Using Reveal ...................................... 20
1.6 Online Help ................................................................. 21
   1.6.1 Using the CrayPat `pat_help` System ............................... 23
   1.6.2 Using the Cray Apprentice2 Help System .......................... 23
   1.6.3 Using the Reveal Help System ...................................... 23
1.7 Reference Files .............................................................. 24
1.8 Upgrading from Earlier Versions .......................................... 24

## Using `pat_build` [2]

2.1 Basic Profiling .............................................................. 25
   2.1.1 Using Automatic Profiling Analysis ................................ 25
2.2 Using Predefined Trace Groups ............................................ 27
## Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.1.2 -O mpi_rank_order</td>
<td>65</td>
</tr>
<tr>
<td>4.5.1.3 -O mpi_hy_rank_order</td>
<td>66</td>
</tr>
<tr>
<td>4.5.1.4 Observations and Suggestions</td>
<td>66</td>
</tr>
</tbody>
</table>

**Using Cray Apprentice2 [5]**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Launching the Program</td>
<td>69</td>
</tr>
<tr>
<td>5.2 Opening Data Files</td>
<td>70</td>
</tr>
<tr>
<td>5.3 Basic Navigation</td>
<td>70</td>
</tr>
<tr>
<td>5.4 Viewing Reports</td>
<td>71</td>
</tr>
<tr>
<td>5.4.1 Overview Report</td>
<td>71</td>
</tr>
<tr>
<td>5.4.2 Profile Report</td>
<td>73</td>
</tr>
<tr>
<td>5.4.3 Text Report</td>
<td>74</td>
</tr>
<tr>
<td>5.4.4 Environment Reports</td>
<td>74</td>
</tr>
<tr>
<td>5.4.5 Traffic Report</td>
<td>75</td>
</tr>
<tr>
<td>5.4.6 Mosaic Report</td>
<td>75</td>
</tr>
<tr>
<td>5.4.7 Activity Report</td>
<td>76</td>
</tr>
<tr>
<td>5.4.8 Call Tree</td>
<td>76</td>
</tr>
<tr>
<td>5.4.9 I/O Rates</td>
<td>77</td>
</tr>
<tr>
<td>5.4.10 Hardware Reports</td>
<td>78</td>
</tr>
<tr>
<td>5.4.10.1 Hardware Counters Overview Report</td>
<td>78</td>
</tr>
<tr>
<td>5.4.10.2 Hardware Counters Plot</td>
<td>78</td>
</tr>
<tr>
<td>5.4.11 GPU Time Line</td>
<td>78</td>
</tr>
<tr>
<td>5.4.11.1 CPU Call Stack and GPU Stream</td>
<td>79</td>
</tr>
<tr>
<td>5.4.11.2 Histogram</td>
<td>80</td>
</tr>
</tbody>
</table>

**Using Reveal [6]**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Launching the Program</td>
<td>81</td>
</tr>
<tr>
<td>6.2 Using Reveal</td>
<td>82</td>
</tr>
<tr>
<td>6.2.1 Generating Loop Work Estimates</td>
<td>82</td>
</tr>
<tr>
<td>6.2.2 Generating a Program Library</td>
<td>83</td>
</tr>
<tr>
<td>6.2.3 Exploring the Results</td>
<td>83</td>
</tr>
<tr>
<td>6.3 For More Information</td>
<td>83</td>
</tr>
</tbody>
</table>

**Appendix A Environment Variables**

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 pat_build Environment Variables</td>
<td>85</td>
</tr>
<tr>
<td>A.2 Run Time Environment Variables</td>
<td>86</td>
</tr>
<tr>
<td>A.3 pat_report Environment Variables</td>
<td>108</td>
</tr>
</tbody>
</table>
Appendix B  Using the Cray Performance Analysis Tools on Cray XK and Cray XC30 Systems  111

B.1 Module Load Order  .............................................................. 111
B.2 pat_build Differences ............................................................ 112
B.3 Run Time Environment Differences ........................................ 112
B.4 pat_report Differences .......................................................... 113
B.5 Cray XC30 CPU Hardware Counter Differences  ....................... 113
B.6 Cray XC30 CPU Network Counter Differences  ........................ 114

Procedures

Procedure 1. Getting Started with CrayPat-lite .................................. 13
Procedure 2. Using CrayPat-lite ...................................................... 14
Procedure 3. Using CrayPat API Calls ............................................. 36
Procedure 4. Using Automatic Rank Order Analysis  ......................... 64

Tables

Table 1. Run Time Environment Variables Summary  ......................... 43
Table 2. Common Panel Actions .................................................... 71
The Cray Performance Measurement and Analysis Tools (or *CrayPat*) are a suite of optional utilities that enable you to capture and analyze performance data generated during the execution of your program on a Cray system. The information collected and analysis produced by use of these tools can help you to find answers to two fundamental programming questions: *How fast is my program running?* and *How can I make it run faster?*

This guide is intended for programmers and application developers who write, port, or optimize software applications for use on Cray XE, Cray XK, or Cray XC30 systems running the Cray Linux Environment (CLE) operating system. We assume you are already familiar with the application development and execution environments and the general principles of program optimization, and that your application is already debugged and capable of running to planned termination. If you need more information about the application development and debugging environment or the application execution environment, see the *Cray Programming Environment User’s Guide* (S–2529) and *Workload Management and Application Placement for the Cray Linux Environment* (S–2496).

A discussion of massively parallel programming optimization techniques is beyond the scope of this guide.

Cray XE, Cray XK, or Cray XC30 systems feature a variety of processors and support a variety of compilers. Because of this, your results may vary from the examples discussed in this guide.

### 1.1 Understanding Performance Analysis

The performance analysis process consists of three basic steps.

1. **Instrument** your program, to specify what kind of data you want to collect under what conditions.
2. **Execute** your instrumented program, to generate and capture the desired data.
3. **Analyze** the resulting data.
Accordingly, the Cray Performance Measure and Analysis Tools suite consists of the following major components:

- **CrayPat-lite**, a simplified and easy-to-use version of CrayPat that provides basic performance analysis information automatically, with a minimum of user interaction. For more information about using CrayPat-lite, see About CrayPat-lite on page 12.

- **CrayPat**, the full-featured program analysis tool set. CrayPat in turn consists of the following major components.
  - `pat_build`, the utility used to instrument programs
  - the CrayPat run time environment, which collects the specified performance data during program execution
  - `pat_report`, the first-level data analysis tool, used to produce text reports or export data for more sophisticated analysis

- **Cray Apprentice2**, the second-level data analysis tool, used to visualize, manipulate, explore, and compare sets of program performance data in a GUI environment.

- **Reveal**, the next-generation integrated performance analysis and code optimization tool, which enables you to correlate performance data captured during program execution directly to the original source, and identify opportunities for further optimization.

- **Cray PAPI components**, which are support packages for those who want to access hardware performance counters. For more information, see Monitoring Performance Counters on page 52.

All Cray-developed performance analysis tools, including the man pages and help system, are available only when the `perftools` or `perftools-lite` module is loaded, with the exception of the PAPI components, which can also be accessed when the `papi` module is loaded.

**Note:** The `perftools`, `perftools-lite`, and `papi` modules are mutually exclusive. You can have one or another loaded, but not two or three at the same time.

### 1.2 About CrayPat-lite

CrayPat-lite is a simplified, easy-to-use version of the Cray Performance Measurement and Analysis Tool set. CrayPat-lite provides basic performance analysis information automatically, with a minimum of user interaction, and yet offers information useful to users wishing to explore their program's behavior further using the full CrayPat tool set.
Procedure 1. Getting Started with CrayPat-lite

To use CrayPat-lite, follow these steps.

1. Load the perftools-lite module.
   
   % module load perftools-lite

2. Compile, link, and instrument your program.
   
   % make program

3. Run your instrumented program on the Cray system.
   
   % aprun a.out

At the end of your program's normal execution, CrayPat-lite produces the following output:

- A text report to stdout, profiling the program's behavior, identifying where the program spends its execution time, and offering recommendations for further analysis and possible optimizations.

- An .rpt file, capturing the same information in a text file.

- An .ap2 file, which can be used to examine the program's behavior more closely using Cray Apprentice2 or pat_report.

- One or more MPICH_RANK_ORDER_FILE files (each with different suffixes), containing suggestions for optimizing MPI rank placement in subsequent program runs. The number and types of files produced is determined by the information captured during program execution. The files can include rank reordering suggestions based on sent message data from MPI functions, time spent in user functions, or a hybrid of the two.

1.2.1 CrayPat-lite Options

CrayPat-lite supports three basic experiments:

- sample_profile — A sampling experiment, which reports execution time, aggregate MFLOP count, the top time-consuming functions and routines, MPI behavior in user functions (if the application is an MPI program), I/O statistics, and generates the data files listed above and (if applicable) MPI rank order suggestions. This is the default experiment.

- event_profile — A tracing experiment, which generates a profile of the top functions traced as well as node observations and possible rank order suggestions.

- gpu — Focuses on the program's use of GPU accelerators.

By default, CrayPat-lite performs the sample_profile experiment. To perform a different experiment, set the CRAYPAT_LITE environment variable to the desired experiment name.
1.2.2 Using CrayPat-lite

A typical CrayPat-lite session follows these steps.

Procedure 2. Using CrayPat-lite

1. Load the perftools-lite module.
   
   `% module load perftools-lite`

2. Compile, link, and instrument your program.

   `% make program`

   Any .o files generated during this step are saved automatically.

3. Run your instrumented program.

   `% aprun a.out`

4. Review the resulting reports from the default sample_profile experiment. When you're ready to continue with another experiment, delete or rename the a.out file.

   `% rm a.out`

   This will force a subsequent make command to relink the program for a new experiment.

5. Set the CRAYPAT_LITE environment variable to the name of the next experiment.

   `% setenv CRAYPAT_LITE event_profile`

6. Rerun make to relink the program.

   `% make program`

   Since your .o files were saved in step 2, this merely relinks your program.

7. Run the program again.

   `% aprun a.out`

8. Review the resulting reports and data files, and determine whether you want to explore your program's behavior further using the full CrayPat tool set or use one of the MPICH_RANK_ORDER_FILE files to create a customized rank placement. (For more information about customized rank placements, see the instructions contained in the MPICH_RANK_ORDER_FILE and the intro_mpi(3) man page.)
1.2.3 Switching from CrayPat-lite to CrayPat

To switch from using CrayPat-lite to using the full CrayPat tool set, unload the perftools-lite module and load the perftools module.

```bash
% module unload perftools-lite
% module load perftools
```

The perftools-lite and perftools modules are mutually exclusive. You can have one or the other loaded, but not both at the same time.

1.2.4 Determining Whether a Binary is Already Instrumented

To determine whether a binary has already been instrumented using CrayPat or CrayPat-lite, use the `strings` command to search for CrayPat/X. For example, if the binary is named `a.out`, use the following command line. If the binary is instrumented, it will return the CrayPat version number and other information.

```bash
% strings a.out | grep 'CrayPat/X'
```

```
CrayPat/X: Version 6.1.2.11786 Revision 11786 09/03/13 00:49:57
```

**Note:** This detects only whether a binary has been instrumented using CrayPat or CrayPat-lite. If the binary has been instrumented using another tool, such as the MPI profiling mechanism, instrumented it again with CrayPat may not succeed, or if it does appear to succeed, the resulting instrumented program may not execute correctly or produce valid results.

1.3 About CrayPat

To use the Cray Performance Measurement and Analysis Tools, first load your programming environment of choice (including CPU or other targeting modules as required), and then load the perftools module.

```
> module load perftools
```

For successful results, the perftools module must be loaded before you compile the program to be instrumented, instrument the program, execute the instrumented program, or generate a report. If you want to instrument a program that was compiled before the perftools module was loaded, you may under some circumstances find that relinking it is sufficient, but as a rule it's best to load the perftools module and then recompile.
When instrumenting a program, CrayPat requires that the object (.o) files created during compilation be present, as well as the library (.a) files, if any. However, most compilers automatically delete the .o and .a files when working with single source files and compiling and linking in a single step, therefore it is good practice to compile and link in separate steps and use the compiler command line option to preserve these files. For example, if you are using the Cray Compiling Environment (CCE) Fortran compiler, compile using either of these command line options:

>`ftn -c sourcefile.f`

Alternatively:

>`ftn -h keepfiles sourcefile.f`

Then link the object files to create the executable program:

>`ftn -o executable sourcefile.o`

For more information about compiling and linking, see your compiler's documentation.

### 1.3.1 Instrumenting the Program

After the `perftools` module is loaded and the program is compiled and linked, you can instrument your program for performance analysis experiments. This is done using the `pat_build` command. In simplest form, it is used like this:

>`pat_build executable`

This produces a copy of your original program, which is named `executable+pat` (for example, `a.out+pat`) and instrumented for the default experiment. Your original executable remains untouched.

The `pat_build` command supports a large number of options and directives, including an API that enables you to instrument specified regions of your code. These options and directives are documented in the `pat_build(1)` man page and discussed in Chapter 2, Using `pat_build` on page 25.

The CrayPat API is discussed in Advanced Users: The CrayPat API on page 36.

#### 1.3.1.1 Automatic Profiling Analysis

The default experiment is Automatic Profiling Analysis, which is an automated process for determining which `pat_build` options are mostly likely to produce meaningful data from your program. For more information about using Automatic Profiling Analysis, see Using Automatic Profiling Analysis on page 25.
1.3.1.2 MPI Automatic Rank Order Analysis

CrayPat is also capable of performing Automatic Rank Order Analysis on MPI programs, and of generating a suggested rank order list for use with MPI rank placement options. Use of this feature requires that the program be instrumented in `pat_build` using either the `-g mpi` or `-O apa` option. For more information about using MPI Automatic Rank Order Analysis, see MPI Automatic Rank Order Analysis on page 64.

1.3.2 Running the Program and Collecting Data

Instrumented programs are executed in exactly the same way as any other program; either by using the `aprun` command if your site permits interactive sessions or by using your system's batch commands.

When working on a Cray system, always pay attention to your file system mount points. While it may be possible to execute a program on a login node or while mounted on the `ufs` file system, this generally does not produce meaningful data. Instead, always run instrumented programs on compute nodes and while mounted on a high-performance file system that supports record locking, such as the Lustre file system.

CrayPat supports more than fifty optional run time environment variables that enable you to control instrumented program behavior and data collection during execution. For example, if you use the C shell and want to collect data in detail rather than in aggregate, consider setting the `PAT_RT_SUMMARY` environment variable to 0 (off) before launching your program.

```
/lus/nid00008> setenv PAT_RT_SUMMARY 0
```

*Note:* Switching off data summarization will record detailed data with timestamps, which can nearly double the number of reports available in Cray Apprentice2, but at the cost of potentially enormous raw data files and significantly increased overhead.

The CrayPat run time environment variables are documented in the `intro_craypat(1)` man page and discussed in Chapter 3, Using the CrayPat Run Time Environment on page 43. The full set of CrayPat run time environment variables is listed in Run Time Environment Variables on page 86.

1.3.3 Analyzing the Results

Assuming your instrumented program runs to completion or planned termination, CrayPat outputs one or more data files. The exact number, location, and content of the data file(s) will vary depending on the nature of your program, the type of experiment for which it was instrumented, and the run time environment variable settings in effect at the time of program execution.
All initial data files are output in .xf format, with a generated file name consisting of your original program name, plus pat, plus the execution process ID number, plus the execution node number. Depending on the program run and the types of data collected, CrayPat output may consist of either a single .xf data file or a directory containing multiple .xf data files.

**Note:** When working with dynamically linked programs, it is recommended that pat_report be invoked on the .xf file promptly after the completion of program execution, in order to produce an .ap2 file. This ensures that the mapping of addresses in dynamic libraries to function names will use the same versions of those libraries that were used when the program was run.

### 1.3.3.1 Initial Analysis: Using pat_report

To begin analyzing the captured data, use the pat_report command. In simplest form, it looks like this:

```
/lus/nid00008> pat_report myprog+pat+PID-node.xf
```

The pat_report command accepts either a file or directory name as input and processes the .xf file(s) to generate a text report. In addition, it also exports the .xf data to a single .ap2 file, which is both a self-contained archive that can be reopened later using the pat_report command and the exported-data file format used by Cray Apprentice2.

The pat_report command provides more than thirty predefined report templates, as well as a large variety of user-configurable options. These reports and options are documented in the pat_report(1) man page and discussed in Chapter 4, Using pat_report on page 57.

**Note:** If you are upgrading from an earlier version of CrayPat, see Upgrading from Earlier Versions on page 24 for important information about data file compatibility.

### 1.4 In-depth Analysis: Using Cray Apprentice2

Cray Apprentice2 is a GUI tool for visualizing and manipulating the performance analysis data captured during program execution. After you use pat_report to open the initial .xf data file(s) and generate an .ap2 file, use Cray Apprentice2 to open and explore the .ap2 file in further detail.

Cray Apprentice2 can display a wide variety of reports and graphs, depending on the type of program being analyzed and the data collected during program execution. The number and appearance of the reports generated using Cray Apprentice2 is determined by the kind and quantity of data captured during program execution, which in turn is determined by the way in which the program was instrumented and the environment variables in effect at the time of program execution.
Cray Apprentice2 is not integrated with CrayPat. You do not set up or run performance analysis experiments from within Cray Apprentice2, nor can you launch Cray Apprentice2 from within CrayPat. Rather, use `pat_build` first, to instrument your program and capture performance data; then use `pat_report` to process the raw data and convert it to `.ap2` format; and then use Cray Apprentice2, to visualize and explore the resulting data files.

Feel free to experiment with the Cray Apprentice2 user interface, and to left- or right-click on any area that looks like it might be interesting. Because Cray Apprentice2 does not write any data files*, you cannot corrupt, truncate, or otherwise damage your original experiment data using Cray Apprentice2.

**Note:** However, under some circumstances it is possible to use the Cray Apprentice2 text report to overwrite generated `MPICH_RANK_ORDER` files. If this happens, you can use `pat_report` to regenerate the rank order files from the original `.xf` data files, if desired. For more information, see MPI Automatic Rank Order Analysis on page 64.

### 1.4.1 On Linux Systems

To begin using Cray Apprentice2 on the Cray system or on a standalone Linux system, verify that the `perftools` module is loaded:

```bash
> module load perftools
```

Launch the Cray Apprentice2 application using the `app2` command:

```bash
> app2 &
```

**Note:** Cray Apprentice2 requires that your workstation be configured to host X Window System sessions. If the `app2` command returns an "cannot open display" error, contact your system administrator for help in configuring X Window System hosting.

You can specify an `.ap2` data file to be opened when you launch Cray Apprentice2:

```bash
> app2 my_datafile.ap2 &
```

Otherwise, Cray Apprentice2 opens a file selection window and you can then select the file you want to open.

For more information about using the `app2` command, see the `app2(1)` man page.
1.4.2 On Microsoft Windows 7 Systems

The optional Windows 7 version of Cray Apprentice2 is launched just like any other Windows program. Double-click on the Cray Apprentice2 icon, and then use the file selection window to navigate to and select the data file you want to open. Alternatively, you can double-click on an .ap2 data file to launch Cray Apprentice2 and open that data file.

Note: The Windows version of Cray Apprentice2 is supported on Microsoft Windows 7 only. It does not work on earlier versions of the Windows operating system and is untested on Microsoft Windows 8 at this time.

1.4.3 On Apple Macintosh Systems

The optional Apple Macintosh version of Cray Apprentice2 is launched just like any other Macintosh program. Double-click on the Cray Apprentice2 icon, and then use the file selection window to navigate to and select the data file you want to open. Alternatively, you can double-click on an .ap2 data file to launch Cray Apprentice2 and open that data file.

Note: The Macintosh version of Cray Apprentice2 was tested on Mac OS 10.6.8. It does not run on the Apple iPad at this time.

1.5 Source Code Analysis: Using Reveal

Reveal is Cray's next-generation integrated performance analysis and code optimization tool. Reveal extend's Cray's existing performance measurement, analysis, and visualization technology by combining run time performance statistics and program source code visualization with Cray Compiling Environment (CCE) compile-time optimization feedback.

Reveal supports source code navigation using whole-program analysis data and program libraries provided by the Cray Compiling Environment, coupled with performance data collected during program execution by the Cray performance tools, to understand which high-level serial loops could benefit from improved parallelism. Reveal provides enhanced loopmark listing functionality, dependency information for targeted loops, and assists users optimizing code by providing variable scoping feedback and suggested compiler directives.

To begin using Reveal on the Cray system, verify that the perftools module is loaded:

> module load perftools
Launch the Reveal application using the `reveal` command:

```
> reveal
```

**Note:** Reveal requires that your workstation be configured to host X Window System sessions. If the `reveal` command returns an "cannot open display" error, contact your system administrator for help in configuring X Window System hosting.

You can specify data files to be opened when you launch Reveal. For example, this command launches Reveal and opens both the compiler-generated program library file and the CrayPat-generated run time performance data file, thus enabling you to correlate performance data captured during program execution with specific lines and loops in the original source code:

```
> reveal my_program_library.pl my_performance_datafile.ap2
```

Alternately, Reveal opens a file selection window and you can then select the data file(s) you want to open.

For more information about using the `reveal` command, see the `reveal(1)` man page.

### 1.6 Online Help

The CrayPat man pages, online help, and FAQ are available only when the `perftools` or `perftools-lite` modules are loaded.
The CrayPat commands, options, and environment variables are documented in the following man pages:

- `craypat-lite(1)` — basic usage information for CrayPat-lite
- `intro_craypat(1)` — basic usage and environment variables for CrayPat
- `pat_build(1)` — instrumenting options and API usage for CrayPat
- `hwpc(5)` — optional hardware counter groups that can be enabled during program execution for CrayPat
- `nwpc(5)` — optional network performance counters that can be enabled during program execution for CrayPat
- `nbpc(5)` — optional AMD Interlagos Northbridge performance counters that can be enabled during program execution for CrayPat (Cray XE and Cray XK systems only)
- `rapl(5)` — optional Intel Running Average Power Limit (RAPL) performance counters that can be enabled to provide socket-level data during program execution for CrayPat (Cray XC30 and Cray XC30-AC systems only)
- `pmpc(5)` — optional Cray Power Management (PM) performance counters that can be enabled to provide node-level data during program execution for CrayPat (Cray XC30 and Cray XC30-AC systems only)
- `accpc(5)` — optional GPU accelerator hardware performance counters that can be enabled during program execution for CrayPat
- `accpc_k20(5)` — optional hardware performance counters specific to the NVIDIA K20 accelerators for CrayPat
- `accpc_x2090(5)` — optional hardware performance counters specific to the NVIDIA X2090 accelerators for CrayPat
- `pat_report(1)` — reporting and data-export options for CrayPat
- `pat_help(1)` — accessing and navigating the command-line driven online help system CrayPat
- `grid_order(1)` — optional CrayPat standalone utility that can be used to generate MPI rank order placement files (MPI programs only)
- `reveal(1)` — introduction to the Reveal integrated code analysis and optimization assistant
Additional useful information can be found in the following man pages.

- intro_mpi(3) — introduction to the MPI library, including information about using MPICH rank reordering information produced by CrayPat (man page available only when the cray-mpich module is loaded)
- intro_papi(3) — introduction to the PAPI library, including information about using PAPI to address hardware and network program counters
- papi_counters(5) — additional information about PAPI utilities

1.6.1 Using the CrayPat pat_help System

CrayPat includes an extensive command-line driven online help system, which features many examples and the answers to many frequently asked questions. To access the help system, type this command:

> pat_help

The pat_help command accepts options. For example, to jump directly into the FAQ, type this command:

> pat_help FAQ

Once the help system is launched, navigation is by one-key commands (e.g., / to return to the top-level menu) and text menus. It is not necessary to enter entire words to make a selection from a text menu; only the significant letters are required. For example, to select "Building Applications" from the FAQ menu, it is sufficient to enter Build.

Help system usage is documented further in the pat_help(1) man page.

1.6.2 Using the Cray Apprentice2 Help System

Cray Apprentice2 features a GUI Javahelp system as well as numerous pop-ups and tool-tips that are displayed by hovering the cursor over an area of interest on a chart or graph. To access the online help system, click the Help button, or right-click on any report tab and then select Panel Help from the pop-up menu.

1.6.3 Using the Reveal Help System

Reveal features an integrated help system as well as numerous pop-ups and tips that are displayed by hovering the cursor over an area of interest in the source code. To access the integrated help system, click the Help button.
1.7 Reference Files

When the performance tools module is loaded, the environment variable CRAYPAT_ROOT is defined. Advanced users will find the files in $CRAYPAT_ROOT/share and $CRAYPAT_ROOT/include to be useful. The /share directory contains the predefined trace group definitions (see Using Predefined Trace Groups on page 27) and build directives (see Advanced Users: Environment Variables and Build Directives on page 31), while the /include directory contains the files used with the CrayPat API (see Advanced Users: The CrayPat API on page 36).

1.8 Upgrading from Earlier Versions

If you are upgrading from an earlier version of the Cray Performance Analysis Tools suite, note the following issues.

- The module names have been changed. Prior to release 5.1, CrayPat and Cray Apprentice2 were packaged in separate module files named xt-craypat and apprentice2 respectively. Beginning with release 5.1 and continuing through the current release, CrayPat and Cray Apprentice2 are integrated into a single module file named perftools.

- File compatibility is not maintained between versions. Programs instrumented using earlier versions of CrayPat must be recompiled, relinked, and reinstrumented using the current version of CrayPat. Likewise, .xf and .ap2 data files created using earlier versions of CrayPat or Cray Apprentice2 cannot be read using the current release.

- If you have upgraded to release 6.1.x from an earlier version, the earlier version likely remains on your system in the /opt/cray/modulefiles/perftools directory. (This may vary depending on your site's software administration and default version policies.) To revert to the earlier version, you must unload the current perftools module and then load the older module.

For example, to revert from CrayPat 6.1.x to CrayPat 6.0.0 so that you can read an old .ap2 file, enter these commands:

> module unload perftools
> module load perftools/6.0.0

To return to the current default version, reverse the commands:

> module unload perftools/6.0.0
> module load perftools
The `pat_build` command is the instrumenting component of the CrayPat performance analysis tool. After you load the `perftools` module and recompile your program, use the `pat_build` command to instrument your program for data capture.

CrayPat supports two categories of performance analysis experiments: *tracing* experiments, which count some event such as the number of times a specific system call is executed, and asynchronous (*sampling*) experiments, which capture values at specified time intervals or when a specified counter overflows.

The `pat_build` command is documented in more detail in the `pat_build(1)` man page. For additional information and examples, see `pat_help build`.

### 2.1 Basic Profiling

The easiest way to use the `pat_build` command is by accepting the defaults.

```bash
> pat_build myprogram
```

This generates a copy of your original executable that is instrumented for the default experiment, Automatic Profiling Analysis. A variety of other predefined experiments are available (see Selecting a Predefined Experiment on page 49), however, Automatic Profiling Analysis is usually the best place to start.

#### 2.1.1 Using Automatic Profiling Analysis

The Automatic Profiling Analysis feature lets CrayPat suggest how your program should be instrumented, in order to capture the most useful data from the most interesting areas. To use this feature, follow these steps.

1. Instrument the original program.

   ```bash
   $ pat_build my_program
   ```

   This produces the instrumented executable `my_program+pat`.

2. Run the instrumented executable.

   ```bash
   $ aprun my_program+pat
   ```

   This produces the data file `my_program+pat+PID-node.t.xf`, which contains basic asynchronously derived program profiling data.
3. Use `pat_report` to process the data file.

$ `pat_report my_program+pat+PID-nodet.xf`

This produces three results:

- a sampling-based text report to `stdout`
- an `.ap2` file (`my_program+pat+PID-nodet.ap2`), which contains both the report data and the associated mapping from addresses to functions and source line numbers
- an `.apa` file (`my_program+pat+PID-nodet.apa`), which contains the `pat_build` arguments recommended for further performance analysis

4. Reinstrument the program, this time using the `.apa` file.

$ `pat_build -O my_program+pat+PID-nodet.apa`

It is not necessary to specify the program name, as this is specified in the `.apa` file. Invoking this command produces the new executable, `my_program+apa`, this time instrumented for enhanced tracing analysis.

5. Run the new instrumented executable.

$ `aprun my_program+apa`

This produces the new data file `my_program+pat+PID2-nodet.xf`, which contains expanded information tracing the most significant functions in the program.

6. Use `pat_report` to process the new data file.

$ `pat_report my_program+pat+PID2-nodet.xf`

This produces two results:

- a tracing report to `stdout`
- an `.ap2` file (`my_program+pat+PID2-nodet.ap2`) containing both the report data and the associated mapping from addresses to functions and source line numbers

When certain conditions are met (job size, data availability, etc.), `pat_report` also attempts to detect a grid topology and evaluate alternative rank orders for opportunities to minimize off-node message traffic, while also trying to balance user time across the cores within a node. These rank-order observations appear on the profile report, and depending on the results, `pat_report` may also generate one or more `MPICH_RANK_ORDER` files for use with the `MPICH_RANK_REORDER_METHOD` environment variable in subsequent application runs.

For more information about MPI rank order analysis, see MPI Automatic Rank Order Analysis on page 64.
For more information about Automatic Profiling Analysis, see the APA topic in pat_help.

2.2 Using Predefined Trace Groups

After Automatic Profiling Analysis, the next-easiest way to instrument your program for tracing is by using the \(-g\) option to specify a predefined trace group.

\>
\texttt{pat\_build \ -g \ tracegroup \ myprogram}

These trace groups instrument the program to trace all function references belonging to the specified group. Only those functions actually executed by the program at run time are traced. \texttt{tracegroup} is case-insensitive and can be one or more of the values listed below.

If the \texttt{tracegroup} name is preceded by the \texttt{!} ("bang") character, the functions within the specified \texttt{tracegroup} are \textbf{not} traced.

- \texttt{adios} Adaptable I/O System API
- \texttt{aio} Functions that perform Asynchronous I/O
- \texttt{armci} Aggregate Remote Memory Copy
- \texttt{blacs} Basic Linear Algebra communication subprograms
- \texttt{blas} Basic Linear Algebra subprograms
- \texttt{caf} Co-Array Fortran (Cray CCE compiler only)
- \texttt{chapel} Chapel language compile and run time library API
- \texttt{cuda} NVIDIA Compute Unified Device Architecture runtime and driver API
- \texttt{dl} Functions that perform dynamic linking
- \texttt{dmapp} Distributed Memory Application API for Gemini and Aries
- \texttt{ffio} Functions that perform Flexible File I/O (Cray CCE compiler only)
- \texttt{fftw} Fast Fourier Transform library
- \texttt{ga} Global Arrays API
- \texttt{gni} Generic Network Interface API
- \texttt{hdf5} Hierarchical Data Format library
- \texttt{heap} Dynamic heap
- \texttt{huge} Linux huge pages
- \texttt{io} Functions and system calls that perform I/O
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lapack</td>
<td>Linear Algebra Package</td>
</tr>
<tr>
<td>math</td>
<td>POSIX.1 math definitions</td>
</tr>
<tr>
<td>mpi</td>
<td>MPI</td>
</tr>
<tr>
<td>netcdf</td>
<td>Network common data form (manages array-oriented scientific data)</td>
</tr>
<tr>
<td>oacc</td>
<td>OpenAccelerator API</td>
</tr>
<tr>
<td>omp</td>
<td>OpenMP API</td>
</tr>
<tr>
<td>pblas</td>
<td>Parallel Basic Linear Algebra Subroutines</td>
</tr>
<tr>
<td>petsc</td>
<td>Portable Extensible Toolkit for Scientific Computation (supported for &quot;real&quot; computations only)</td>
</tr>
<tr>
<td>pgas</td>
<td>Parallel Global Address Space</td>
</tr>
<tr>
<td>pthreads</td>
<td>POSIX threads</td>
</tr>
<tr>
<td>realtime</td>
<td>POSIX real time extensions</td>
</tr>
<tr>
<td>scalapack</td>
<td>Scalable LAPACK</td>
</tr>
<tr>
<td>sheap</td>
<td>Shared heap</td>
</tr>
<tr>
<td>shmem</td>
<td>SHMEM</td>
</tr>
<tr>
<td>spawn</td>
<td>POSIX real time process creation</td>
</tr>
<tr>
<td>stdio</td>
<td>All library functions that accept or return the FILE* construct</td>
</tr>
<tr>
<td>string</td>
<td>String operations</td>
</tr>
<tr>
<td>syscall</td>
<td>System calls</td>
</tr>
<tr>
<td>sysfs</td>
<td>System calls that perform miscellaneous file management</td>
</tr>
<tr>
<td>sysio</td>
<td>System calls that perform I/O</td>
</tr>
<tr>
<td>upc</td>
<td>Unified Parallel C (Cray CCE compiler only)</td>
</tr>
</tbody>
</table>
The files that define the predefined trace groups are kept in $CRAYPAT_ROOT/share. To see exactly which functions are being traced in any given group, examine the Trace files. These files can also be used as templates for creating user-defined tracing files. (See Instrumenting a User-defined List of Functions on page 30.)

Note: There is a dependency between the way in which a program is instrumented using pat_build and the information subsequently available for use in pat_report. For example, you must instrument a program to collect MPI information (either by using the -g mpi option listed above or by using one of the user-defined tracing options listed below) in order to see MPI data on any of the reports produced by pat_report. For more information, see Using Predefined Reports on page 59.

2.3 Tracing User-defined Functions

Alternatively, you can use the pat_build command options to instrument specific functions, to instrument a user-defined list of functions, to block the instrumentation of specific functions, or to create new trace intercept routines.

2.3.1 Enabling Tracing and the CrayPat API

To change the default experiment from Automatic Profiling Analysis to tracing, activate any API calls added to your program, and enable tracing for user-defined functions, use the -w option.

> pat_build -w myprogram

The -w option has other implications which are discussed in the following sections.

2.3.2 Instrumenting a Single Function

To instrument a specific function by name, use the -T option.

> pat_build -T tracefunc myprogram

If tracefunc is a user-defined function, the -w option must also be specified in order to create a trace wrapper for the function. (See Using Predefined Trace Groups on page 27.) If the -w option is not specified, only those function entry points that have predefined trace intercept routines are traced.

If tracefunc contains a slash (/) character, the string is interpreted as a basic regular expression. If more than one regular expression is specified, the union of all regular expressions is taken. All functions that match at least one of the regular expressions are added to the list of functions to trace. The match is case-sensitive. For more information about UNIX regular expressions, see the regexec(3) man page.
One or more regular expression qualifiers can precede the slash (/) character. The ! qualifier means reverse the results of the match, the i qualifier means ignore case when matching, and the x qualifier means use extended regular expressions.

### 2.3.3 Preventing Instrumentation of a Function

To prevent instrumentation of a specific function, use the ~T ! option.

```bash
> pat_build ~T !tracefunc myprogram
```

If `tracefunc` begins with an exclamation point (!) character, references to `tracefunc` are not traced.

### 2.3.4 Instrumenting a User-defined List of Functions

To trace a user-defined list of functions, use the -t option.

```bash
> pat_build -t tracefile myprogram
```

The `tracefile` is a plain ASCII text file listing the functions to be traced. For an example of a `tracefile`, see any of the predefined Trace files in `$CRAYPAT_ROOT/share`.

To generate trace wrappers for user-defined functions, also include the -w option. If the -w option is not specified, only those function entry points that have predefined trace intercept routines are traced.

### 2.3.5 Creating New Trace Intercept Routines for User-defined Functions

To create new trace intercept routines for those functions that are defined in the respective source file owned by the user, use the -u option.

```bash
> pat_build -u myprogram
```

To prevent a specific function entry-point from being traced, use the ~T ! option.

```bash
> pat_build -u ~T 'entry-point' myprogram
```

### 2.3.6 Creating New Trace Intercept Routines for Everything

To make tracing the default experiment, activate the CrayPat API, and create new trace intercept routines for those functions for which no trace intercept routine already exists, use the -w option.

```bash
> pat_build -w -t tracefile[...] ~T symbol[...] myprogram
```
If -t, -T, or the trace build directive are not specified, only those functions necessary to support the CrayPat run time library are traced. If -t, -T, or the trace build directive are specified, and -w is not specified, only those function points that have predefined trace intercept routines are traced. If the list of functions to be traced includes any user-defined functions, the -w option must also be specified to generate trace wrappers.

2.4 Advanced Users: Environment Variables and Build Directives

CrayPat supports a number of environment variables and build directives that enable you to fine-tune the behavior of the pat_build command. The following environment variables are currently supported.

**PAT_BUILD_CLEANUP**

By default or if set to a nonzero value, the intermediate directory is removed. Set to zero to retain the directory after pat_build completes.

**PAT_BUILD_EMBED_RTENV**

Specifies one or more comma-separated CrayPat run time environment variables to embed in the instrumented program. The CrayPat run time environment variables must be set at the time the instrumented program is created. By default, all CrayPat run time environment variables that are set at the time the instrumented program is created are embedded in the instrumented program. If **PAT_BUILD_EMBED_RTENV** is set to zero, no CrayPat run time environment variables are embedded. For more information, see the description of the rtenv build directive later in this section.

**PAT_BUILD_OPTIONS**

If set, specifies the pat_build options that are to be evaluated before any options on the command line.

**PAT_BUILD_PAPI_DIR**

Specifies the location of the PAPI run time library. If this environment variable is set, the directory path is valid, and the libpapi.a or libpapi.so file exists in the specified directory, the respective file is used to satisfy the PAPI requirements of the CrayPat run time library when the instrumented program is created.

Default: /opt/cray/papi/current_version
PAT_BUILD_PRMGT

If set to nonzero, forces Process Management (PMI) entry points to be loaded into the instrumented program. If set to zero, no additional PMI entry points are loaded into the instrumented program. If not set (default), PMI entry points are loaded into the instrumented program only if the programming models present in the input program dictate the need for PMI.

PAT_BUILD_VERBOSE

If set, specifies the detail level of the progress messages related to the instrumentation process. This value corresponds to the number of -v options specified.

PAT_LD_OBJECT_TMPDIR

Allows the user to change the location of the directory where CrayPat copies of object files that are in a /tmp directory. When set, pat_build writes copies of object files into the $PAT_LD_OBJECT_TMPDIR/.craypat/program-name/PID-of-link directory. The default value for PAT_LD_OBJECT_TMPDIR is $HOME.

To disable copying of object files, set this environment variable to 0 (zero).

Build directives are invoked either by using the pat_build -d option to read in a build directives file (by default, $CRAYPAT_ROOT/share/BuildDirectives), or by using the pat_build -D option to specify individual directives. The following build directives are currently supported. The format of each directive is directivename=directivevalue.

addsym-archive=y | n

If set to y archive files writable by the user are eligible to have their functions traced when the -u option is specified. This is the default behavior.

addsym-weak=y | n

If set to y functions defined with WEAK binding in files writable by the user are eligible to have their functions traced when the -u option is specified. This is the default behavior.
force-instr=y|n

By default, the pat_build command does not permit a program to be instrumented if it already has been instrumented by another method. If this directive is set to y, the pat_build command ignores the check for prior instrumentation and attempts to force instrumentation of the program. The other methods of instrumenting a program include:

- the PERFCTR, PFM, or PAPI libraries
- the IOBUF or FPMPI libraries
- GNU profiling or GNU coverage analysis
- MPI profiling functions
- previous use of the pat_build command

⚠️ **Caution:** Using this directive to force instrumentation of a previously instrumented program may result in an executable that produces incorrect results, exhibits unpredictable behavior, or generates invalid CrayPat performance analysis data.

invalid=entry-point[, entry-point...]

Specifies one or more functions in the original program that inhibit any instrumentation.

link-fatal=operand[, operand...]

Specifies one or more operands that, if present in the original link, will prevent the instrumented link from occurring.

link-ignore=operand[, operand...]

Specifies one or more operands that, if present in the original link, will not be passed down to the instrumented link.

link-ignore-libs=lib[, lib...]

Specifies one or more object or archive files that, if present in the original link, will not be passed down to the instrumented link.

link-instr=operand[, operand...]

Specifies one or more operands to include in the instrumented link.

link-minus-u=entry-point[, entry-point,...]

Adds the ld -u option for each entry-point to the relink command, forcing the entry-point to be loaded into the instrumented executable.

link-objs=ofile[, ofile...]

Specifies one or more object files to include in the instrumented link.
link-symbol=entry-point[, entry-point,...]

Adds the ld -y option for each entry-point to the relink command, showing where the entry-point is being referenced and from where it is resolved.

report=y|n
Generates a text report upon completion of the instrumented program's successful execution. The report is written to stdout. The default is n.

rtenv=name=value[:name=value;...]

Embeds the run time environment variable name in the instrumented program and sets it to value. If a run time environment variable is set using both this directive and in the execution environment, the value set in the execution environment takes precedence and this value is ignored.

For more information about run time environment variables, see the intro_craypat(1) man page.

trace=entry-point[, entry-point,...]

Specifies one or more functions in the original program to trace. If entry-point is preceded by the ! character, function entry-point is not allowed to be traced.

trace-args=y | n

Collect and record at run time the values of formal parameters for generated trace intercept routines. The default is n.

trace-complex=y | n

If set to y, generate a wrapper for functions that return a complex value. The default is n.

trace-debug=strng[strng2,...]

Add verbose print statements to generated trace intercept routines. The string strng identifies part or all of the function name. The print statements are activated at run time when the PAT_RT_VERBOSE environment variable is set to nonzero. This may be helpful if a traced function is suspected of causing a run time error.
trace-file=\textit{strng}[,\textit{strng2},...]

Activate or deactivate tracing of functions in a file. The string \textit{strng} identifies part or all of the file name to activate or deactivate. If \textit{strng} is preceded by an exclamation point (!) functions in the matched file(s) are not traced.

\textbf{trace-max}=n

The maximum number of functions in the original program that can be traced based on the size of the function. By default, the 1024 largest functions by size are traced. If \textit{n} is less than zero, the \textit{n} smallest functions are traced. This directive can be combined with the \texttt{trace-text-size} directive to trace a limited number of functions within a specified range. Tracing a large number of functions results in degraded performance of the instrumented program at run time.

\textbf{trace-return-size}=\textit{min},\textit{max}

Specifies the minimum and maximum size in bytes of the return value of a user function to trace. User functions that return fewer than \textit{min} bytes or greater than \textit{max} bytes are not traced.

Defaults: 0,16

\textbf{trace-skip}=\textit{strng}[,\textit{strng2},...]

Silently ignore functions when processing them for tracing. The string \textit{strng} identifies part or all of the function name.

\textbf{trace-text-size}=\textit{min},\textit{max}

Specifies the minimum and maximum size in bytes of text sections in user-defined functions to trace. This does not apply to functions defined in the trace function groups.

\textbf{varargs}=y | n

If set to \textit{y}, functions that accept variable arguments can be traced. The default is \textit{n}.
2.5 Advanced Users: The CrayPat API

There may be times when you want to focus on a certain region within your code, either to reduce sampling overhead, reduce data file size, or because only a particular region or function is of interest. In these cases, use the CrayPat API to insert calls into your program source, to turn data capture on and off at key points during program execution. By using the CrayPat API, it is possible to collect data for specific functions upon entry into and exit from the functions, or even from one or more regions within the body of the function.

Procedure 3. Using CrayPat API Calls

1. Load the performance tools module.
   
   > module load perftools

2. Include the CrayPat API header file in your source code. Header files for both Fortran and C/C++ are provided in $CRAYPAT_ROOT/include.

3. Modify your source code to insert API calls where wanted.

4. Compile your code.

   Use the pat_build -w option to build the instrumented executable.
   Additional functions can also be specified using the -t or -T options. The -u option (see Creating New Trace Intercept Routines for User-defined Functions on page 30) may be used, but it is not recommended as it forces pat_build to create an entry point for every user-defined function, which may inject excessive tracing overhead and obscure the results for the regions.

5. Run the instrumented program, and use the pat_report command to examine the results.

2.5.1 Header Files

CrayPat API calls are supported in both Fortran and C. The include files are found in $CRAYPAT_ROOT/include.

The C header file, pat_api.h, must be included in your C source code.

The Fortran header files, pat_apif.h and pat_apif77.h, provide important declarations and constants and should be included in those Fortran source files that reference the CrayPat API. The header file pat_apif.h is used only with compilers that accept Fortran 90 constructs such as new-style declarations and interface blocks. The alternative Fortran header file, pat_apif77.h, is for use with compilers that do not accept such constructs.
When the perftools module is loaded it defines a compiler macro called CRAYPAT. This macro can be useful when adding any of the following API calls or include statements to your program to make them conditional:

```c
#if defined(CRAYPAT)
 <function call>
#endif
```

This macro may be activated manually by compiling with the `-D CRAYPAT` argument or otherwise defined by using the `#define` preprocessor macro.

### 2.5.2 API Calls

The following API calls are supported. All API usage must begin with a `PAT_region_begin` call and end with a `PAT_region_end` call. The examples below show C syntax. The Fortran functions are similar.

```c
PAT_region_begin(int id, const char *label)

PAT_region_end(int id)
```

Defines the boundaries of a region. A region must consist of a sequence of executable statements within a single function, and must have a single entry at the top and a single exit at the bottom. Regions must be either separate or nested: if two regions are not disjoint, then one must entirely contain the other. A region may contain function calls. (These restrictions are similar to the restrictions on an OpenMP structured block.)

For each region, a summary of activity including time and hardware performance counters (if selected) is produced. The argument `id` assigns a numerical value to the region and must be greater than zero. Each `id` must be unique across the entire program.

The argument `label` assigns a character string to the region, allowing for easier identification of the region in the report.

These functions return `PAT_API_OK` if the region request was valid and `PAT_API_FAIL` if the request was not valid.

Two runtime environment variables affect region processing: `PAT_RT_REGION_CALLSTACK` and `PAT_RT_REGION_MAX`. See the `intro_craypat(1)` man page for more information.

```c
PAT_record(int state)
```

`PAT_record` controls the state for all threads on the executing PE.
The PAT_record function sets the recording state to one of the following values and returns the previous state before the call was made.

**PAT_STATE_ON**

If called from the main thread, switches recording on for all threads on the executing PE. Otherwise, switches recording on for just the calling child thread.

**PAT_THREAD_STATE_ON**

Switches recording on for only the thread making the API call.

*Deprecated:* Function scheduled to be removed in the next release.

**PAT_STATE_OFF**

If called from the main thread, switches recording off for all threads on the executing PE. Otherwise, switches recording off for just the calling child thread.

**PAT_THREAD_STATE_OFF**

Switches recording off for only the thread making the API call.

*Deprecated:* Function scheduled to be removed in the next release.

**PAT_STATE_QUERY**

If called from the main thread, returns the state of the main thread on the executing PE. Otherwise, returns the state of the calling child thread.

**PAT_THREAD_STATE_QUERY**

Returns the state of only the thread making the API call.

*Deprecated:* Function scheduled to be removed in the next release.

All other values have no effect on the state.
PAT_trace_user_l(const char *str, int expr, ...)  
Issues a TRACE_USER record into the experiment data file if the expression \textit{expr} evaluates to true. The record contains the identifying string \textit{str} and the arguments, if specified, in addition to other information, including a timestamp.

Returns the value of \textit{expr}.

This function applies to tracing experiments only.

This function is supported for C and C++ programs only, and is not available in Fortran.

PAT_trace_user_v(const char *str, int expr, int nargs, long *args)  
Issues a TRACE_USER record into the experiment data file if the expression \textit{expr} evaluates to true. The record contains the identifying string \textit{str} and the arguments, if specified, in addition to other information, including a timestamp.

\textit{nargs} indicates the number of 64-bit arguments pointed to by \textit{args}. These arguments are included in the TRACE_USER record.

Returns the value of \textit{expr}.

This function applies to tracing experiments only.

PAT_trace_user(const char *str)  
Issues a TRACE_USER record containing the identifying string \textit{str} into the experiment data file. Returns \texttt{PAT\_API\_OK} if the trace record is written to the experiment data file successfully, otherwise, \texttt{PAT\_API\_FAIL} is returned.

This function applies to tracing experiments only.

PAT_trace_function(const void *addr, int state)  
Activates or deactivates the tracing of the instrumented function indicated by the function entry address \textit{addr}. The argument \textit{state} is the same as state above. Returns \texttt{PAT\_API\_OK} if the function at the entry address was activated or deactivated, otherwise, \texttt{PAT\_API\_FAIL} is returned.

This function applies to tracing experiments only.

PAT_flush_buffer(unsigned long *nbytes)  
Writes all the recorded contents in the data buffer to the experiment data file for the calling PE and calling thread. The number of bytes written to the experiment data file is returned in the variable pointed
to by *nbytes. Returns PAT_API_OK if all buffered data was written to the data file successfully, otherwise, returns PAT_API_FAIL. After writing the contents, the data buffer is empty and begins to refill. See intro_craypat(1) to control the size of the write buffer.

PAT_counters(int category, const char *names[], unsigned long values[], int *nevents)

PAT_counters returns the names and current count value of any counter events that have been set to count on the hardware category. The names of these events are returned in the names array of strings, the number of names is returned in the location pointed by to nevents, and the counts are returned for the thread from which the function is called. The values for these events are returned in the values array of integers, and the number of values is returned in the location pointed by to nevents. If both names and values are set to zero then what nevents points to is set to the number of events. The function returns PAT_API_OK if all the event names were returned successfully and PAT_API_FAIL if they were not.

The values for category are:

PAT_CTRS_CPU

Performance counters that reside on the CPU

PAT_CTRS_NETWORK

Performance counters that reside on the network router

PAT_CTRS_ACCEL

Performance counters that reside on any GPU accelerator

PAT_CTRS_NB

Performance counters that reside on the AMD Interlagos Northbridge communication packet routing block.

PAT_CTRS_RAPL

Performance measurements that represent the Intel Running Average Power Level.

PAT_CTRS_PM

Performance measurements that represent Cray Power and Energy Management.
To get just the number of events returned, set names or values to zero.

The event names to be returned are selected at run time using the PAT_RT_PERFCTR environment variable. If no event names are specified, the value of nevents is zero.

**Note:** The data collected by the PAT_trace_user API functions is not currently shown on any report. Advanced users may want to collect it and extract information from a text dump of the data files.

For more information about CrayPat API usage, see the pat_build(1) man page and the APA topic in pat_help.

## 2.6 Advanced Users: OpenMP

For programs that use the OpenMP programming model, CrayPat can measure the overhead incurred by entering and leaving parallel regions and work-sharing constructs within parallel regions, show per-thread timings and other data, and calculate the load balance across threads for such constructs.

For programs that use both MPI and OpenMP, profiles by default show the load balance over PEs of the average time in the threads for each PE, but you can also see load balances for each programming model separately. For more information about reporting load balance by programming model, see the pat_report(1) man page.

The Cray CCE compiler automatically inserts calls to trace points in the CrayPat runtime library in order to support the required CrayPat measurements.

PGI compiler release 7.2.0 or later automatically inserts calls to trace points. For all other compilers, including earlier releases of the PGI compiler suite, the user is responsible for inserting API calls.

The following C functions are used to instrument OpenMP constructs for compilers that do not support automatic instrumentation. Fortran subroutines with the same names are also available.

```c
void PAT_omp_parallel_enter (void);
void PAT_omp_parallel_exit (void);
void PAT_omp_parallel_begin (void);
void PAT_omp_parallel_end (void);
void PAT_omp_loop_enter (void);
void PAT_omp_loop_exit (void);
void PAT_omp_sections_enter (void);
void PAT_omp_sections_exit (void);
void PAT_omp_section_begin (void);
void PAT_omp_section_end (void);
```

Note that the CrayPat OpenMP API does not support combined parallel work-sharing constructs. To instrument such a construct, it must be split into a parallel construct containing a work-sharing construct.
Use of the CrayPat OpenMP API function must satisfy the following requirements.

- If one member of an _enter/_exit or _begin/_end pair is called, the other must also be called.

- Calls to _enter or _begin functions must immediately precede the relevant construct. Calls to _end or _exit functions must immediately follow the relevant construct.

- For a given parallel region, all or none of the four functions with prefix PAT_omp_parallel must be called.

- For a given "sections" construct, all or none of the four functions with prefix PAT_omp_section must be called.

- A "single" construct should be treated as if it were a "sections" construct consisting of one section.
Using the CrayPat Run Time Environment  [3]

The CrayPat run time environment variables communicate directly with an executing instrumented program and affect how data is collected and saved. Detailed descriptions of all run time environment variables are provided in the intro_craypat(1) man page, and in this guide in Run Time Environment Variables on page 86. Additional information can be found in the online help system under pat_help environment.

This chapter provides a summary of the run time environment variables, and highlights some of the more commonly used ones and what they are used for.

3.1 Summary

All CrayPat run time environment variable names begin with PAT_RT_. Some require discrete values, while others are toggles. In the case of all toggles, a value of 1 is on (enabled) and 0 is off (disabled).

Table 1. Run Time Environment Variables Summary

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Short Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAT_RT_ACC_ACTIVITY_BUFFER_SIZE</td>
<td>Specify the size in bytes of the buffer used to collect records for the accelerator time line view.</td>
<td>1 MB</td>
</tr>
<tr>
<td>PAT_RT_ACC_FORCE_SYNC</td>
<td>Toggle: force accelerator synchronization in order to enable collection of accelerator time for asynchronous events. (Cray XK only)</td>
<td>0</td>
</tr>
<tr>
<td>PAT_RT_ACC_RECORD</td>
<td>Overrides the programming model for which performance data is collected.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_BUILD_ENV</td>
<td>Toggle: use runtime environment variables embedded using the pat_build rtenv directive.</td>
<td>1</td>
</tr>
<tr>
<td>PAT_RT_CALLSTACK</td>
<td>Specify the depth to which to trace call stacks.</td>
<td>100</td>
</tr>
<tr>
<td>Variable Name</td>
<td>Short Description</td>
<td>Default</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>PAT_RT_CALLSTACK_BUFFER_SIZE</td>
<td>Specify the size in bytes of the runtime summary buffer used to collect function call stacks.</td>
<td>4MB</td>
</tr>
<tr>
<td>PAT_RT_CHECKPOINT</td>
<td>Toggle: enable checkpoint/restart and if enabled set the maximum number of checkpoint states collected.</td>
<td>32</td>
</tr>
<tr>
<td>PAT_RT_COMMENT</td>
<td>Specify string to insert into experiment data files.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_CONFIG_FILE</td>
<td>Specify configuration file(s) containing runtime environment variables.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_EXIT_AFTER_INIT</td>
<td>Toggle: terminate execution after initialization of the CrayPat runtime library.</td>
<td>0</td>
</tr>
<tr>
<td>PAT_RT_EXPERIMENT</td>
<td>Specify the performance analysis experiment to perform.</td>
<td>Automatic Profiling Analysis</td>
</tr>
<tr>
<td>PAT_RT_EXPFILE_APPEND</td>
<td>Toggle: append experiment data records to existing experiment data file.</td>
<td>0</td>
</tr>
<tr>
<td>PAT_RT_EXPFILE_DIR</td>
<td>Specify the directory in which to write the experiment data file.</td>
<td>current execution directory</td>
</tr>
<tr>
<td>PAT_RT_EXPFILE_FIFO</td>
<td>Toggle: create data file as named FIFO pipe instead of a regular file.</td>
<td>0</td>
</tr>
<tr>
<td>PAT_RT_EXPFILE_FSLOCK</td>
<td>Override the system-determined record-locking attribute.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_EXPFILE_MAX</td>
<td>Specify the maximum number of data files created.</td>
<td>256</td>
</tr>
<tr>
<td>PAT_RT_EXPFILE_NAME</td>
<td>Replaces the name portion of the experiment data file that was appended to the directory.</td>
<td>the base file name</td>
</tr>
<tr>
<td>PAT_RT_EXPFILE_PES</td>
<td>Specify the individual PEs from which to collect and record data.</td>
<td>all PEs</td>
</tr>
<tr>
<td>PAT_RT_EXPFILE_REPLACE</td>
<td>Toggle: enable overwriting of existing experiment data file(s).</td>
<td>0</td>
</tr>
<tr>
<td>PAT_RT_EXPFILE_SUFFIX</td>
<td>Specify the default experiment data filename suffix.</td>
<td>.xf</td>
</tr>
<tr>
<td>PAT_RT_EXPFILE_THREADS</td>
<td>Specify the individual threads from which to collect data.</td>
<td>all threads</td>
</tr>
<tr>
<td>Variable Name</td>
<td>Short Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>PAT_RT_HEAP_BUFFER_SIZE</td>
<td>Specify the size in bytes of the buffer used to collect dynamic heap information.</td>
<td>2MB</td>
</tr>
<tr>
<td>PAT_RT_INTERVAL</td>
<td>Specify the sampling interval in microseconds.</td>
<td>10000</td>
</tr>
<tr>
<td>PAT_RT_INTERVAL_TIMER</td>
<td>Specify the type of POSIX interval timer used for sampling-by-time experiments.</td>
<td>1</td>
</tr>
<tr>
<td>PAT_RT_MPI_MSG_BINS</td>
<td>Specify the sizes of the histogram bins used to capture MPI messages.</td>
<td>16, 256, 4kb, 64kb, 1mb, 16mb</td>
</tr>
<tr>
<td>PAT_RT_MPI_SYNC</td>
<td>Toggle: measure MPI load imbalance by measuring the time spent in barrier and sync calls before entering the collective.</td>
<td>1 for tracing experiments, 0 for sampling experiments</td>
</tr>
<tr>
<td>PAT_RT_MPI_THREAD_REQUIRED</td>
<td>Specifies the MPI thread-level support.</td>
<td>As specified by the calling program</td>
</tr>
<tr>
<td>PAT_RT_PARALLEL_MAX</td>
<td>Specifies the maximum number of unique call site entries to collect for OpenMP trace points.</td>
<td>1024</td>
</tr>
<tr>
<td>PAT_RT_PERFCTR</td>
<td>Specify the performance counter events to be counted.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_PERFCTR_FILE</td>
<td>Specify file(s) containing performance counter event specifications.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_PERFCTR_FILE_GROUP</td>
<td>Specify file(s) containing performance counter group definitions.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_RECORD</td>
<td>Specifies the initial data collection and recording state.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_REGION_CALLSTACK</td>
<td>Specify the maximum stack depth for CrayPat API functions PAT_region_begin and PAT_region_end.</td>
<td>128</td>
</tr>
<tr>
<td>PAT_RT_REGION_MAX</td>
<td>Specify the largest numerical ID that may be used as an argument to CrayPat API functions PAT_region_begin and PAT_region_end.</td>
<td>100</td>
</tr>
<tr>
<td>PAT_RT_REPORT_CLEANUP</td>
<td>Specify how temporary files are removed if a runtime report was generated.</td>
<td>fail</td>
</tr>
<tr>
<td>Variable Name</td>
<td>Short Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>PAT_RT_REPORT_CMD</td>
<td>Specify the executable used to generate the on-completion report and the arguments to be passed to it.</td>
<td>pat_report, none</td>
</tr>
<tr>
<td>PAT_RT_REPORT_METHOD</td>
<td>Specify the mechanism used to create the on-completion text report.</td>
<td>pe0</td>
</tr>
<tr>
<td>PAT_RT_SAMPLING_MASK</td>
<td>Specifies a bitmask that is AND’d with the PC address acquired during sampling.</td>
<td>0xfffffffffffff</td>
</tr>
<tr>
<td>PAT_RT_SAMPLING_MODE</td>
<td>Specify the mode (0, 1, or 3) in which trace-enhanced sampling operates.</td>
<td>0</td>
</tr>
<tr>
<td>PAT_RT_SAMPLING_SIGNAL</td>
<td>Specify the signal issued when a POSIX interval timer expires or a hardware counter overflows.</td>
<td>27 (SIGPROF)</td>
</tr>
<tr>
<td>PAT_RT_SETUP_SIGNAL_HANDLERS</td>
<td>Toggle: ignore received signals in order to produce a more accurate traceback.</td>
<td>1</td>
</tr>
<tr>
<td>PAT_RT_SUMMARY</td>
<td>Toggle: enable runtime summarization and data aggregation.</td>
<td>1</td>
</tr>
<tr>
<td>PAT_RT_THREAD_ALLOW</td>
<td>Specify how created threads are monitored and recorded.</td>
<td>1 (enabled)</td>
</tr>
<tr>
<td>PAT_RT_THREAD_CANCEL_NTRIES</td>
<td>Specify how long to wait for created threads to terminate.</td>
<td>120 (30 seconds)</td>
</tr>
<tr>
<td>PAT_RT_THREAD_MAX</td>
<td>Specify the maximum number of threads that can be created and recorded.</td>
<td>1,000,000</td>
</tr>
<tr>
<td>PAT_RT_TRACE_API</td>
<td>Toggle: enable recording of data generated by CrayPat API functions.</td>
<td>1</td>
</tr>
<tr>
<td>PAT_RT_TRACE_DEPTH</td>
<td>Specify the maximum depth of the runtime callstack.</td>
<td>512</td>
</tr>
<tr>
<td>PAT_RT_TRACE_FUNCTION_ARGS</td>
<td>Specify the maximum number of function argument values recorded each time the function is called.</td>
<td>256</td>
</tr>
<tr>
<td>Variable Name</td>
<td>Short Description</td>
<td>Default</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>PAT_RT_TRACE_FUNCTION_DISPLAY</td>
<td>Toggle: write the function names that have been instrumented to stdout.</td>
<td>0</td>
</tr>
<tr>
<td>PAT_RT_TRACE_FUNCTION_MAX</td>
<td>Set maximum number of traces generated for a single process.</td>
<td>unlimited</td>
</tr>
<tr>
<td>PAT_RT_TRACE_FUNCTION_NAME</td>
<td>Specify by name which instrumented functions to trace in a program instrumented for tracing.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_TRACE_FUNCTION_SIZE</td>
<td>Specify the size of the instrumented function to trace in a program instrumented for tracing.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_TRACE_HEAP</td>
<td>Toggle: collect dynamic heap information.</td>
<td>1</td>
</tr>
<tr>
<td>PAT_RT_TRACE_HOOKS</td>
<td>Toggle: enable/disable recording trace data for specified types of compiler-generated hooks.</td>
<td>depends on PAT_RT_SUMMARY</td>
</tr>
<tr>
<td>PAT_RT_TRACE_OVERHEAD</td>
<td>Specify the number of times calling overhead is sampled during program initialization and termination.</td>
<td>100</td>
</tr>
<tr>
<td>PAT_RT_TRACE_THRESHOLD_PCT</td>
<td>Set relative time threshold below which function trace records are not kept.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_TRACE_THRESHOLD_TIME</td>
<td>Set absolute time threshold below which function trace records are not kept.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_VALIDATE_SYSCALLS</td>
<td>Toggle: prevent program from executing function calls that interfere with data collection.</td>
<td>1</td>
</tr>
<tr>
<td>PAT_RT_VERBOSE</td>
<td>If set, specify the PEs from which to accept and record info-level messages.</td>
<td>unset</td>
</tr>
<tr>
<td>PAT_RT_WRITE_BUFFER_SIZE</td>
<td>Size of single thread data collection buffer in bytes.</td>
<td>8MB</td>
</tr>
</tbody>
</table>

### 3.2 Common Uses

#### 3.2.1 Controlling Run Time Summarization

Environment variable: PAT_RT_SUMMARY
Run time summarization is enabled by default. When it is enabled, data is captured in detail, but automatically aggregated and summarized before being saved. This greatly reduces the size of the resulting experiment data files but at the cost of fine-grain detail. Specifically, when running tracing experiments, the formal parameter values, function return values, and call stack information are not saved.

If you want to study your data in detail, and particularly if you want to use Cray Apprentice2 to generate charts and graphs, disable run time summarization by setting `PAT_RT_SUMMARY` to 0. Doing so can more than double the number of reports available in Cray Apprentice2. However, it does so at the expense of greatly increased data file system and significant execution overhead.

**Note:** Users who use the `PAT_RT_SUMMARY` environment variable to turn off run time summarization often find it helpful to set `PAT_RT_EXPFILE_PES` to 0, in order to reduce redundancy by collecting data only from PE 0.

### 3.2.2 Controlling Data File Size

Depending on the nature of your experiment and the duration of the program run, the data files generated by CrayPat can be quite large. To reduce the files to manageable sizes, considering adjusting the following run time environment variables.

For sampling experiments, try these:

- `PAT_RT_CALLSTACK`
- `PAT_RT_EXPFILE_PES`
- `PAT_RT_INTERVAL`
- `PAT_RT_SAMPLING_MASK`
- `PAT_RT_SUMMARY`

For tracing experiments, try these:

- `PAT_RT_CALLSTACK`
- `PAT_RT_EXPFILE_PES`
- `PAT_RT_SUMMARY`
- `PAT_RT_TRACE_FUNCTION_ARGS`
- `PAT_RT_TRACE_FUNCTION_MAX`
- `PAT_RT_TRACE_FUNCTION_NAME`
- `PAT_RT_TRACE_FUNCTION_SIZE`
- `PAT_RT_TRACE_THRESHOLD_PCT`
- `PAT_RT_TRACE_THRESHOLD_TIME`
Users performing sampling or trace-enhanced sampling experiments on programs running on large numbers of nodes often find it helpful to set \texttt{PAT_RT_INTERVAL} to values larger than the default of 10,000 microseconds. This reduces data granularity, but also reduces the size of the resulting data files.

### 3.2.3 Selecting a Predefined Experiment

Environment variable: \texttt{PAT_RT_EXPERIMENT}

By default, CrayPat instruments programs for Automatic Profiling Analysis. However, if a program is instrumented for a sampling experiment by using the \texttt{pat_build -S} option, or for tracing by using the \texttt{pat_build -w, -U, -u, -T, -t, or -g} options, then you can use the \texttt{PAT_RT_EXPERIMENT} environment variable to further specify the type of experiment to be performed.

The valid experiment types are:

- **samp_pc_time**
  
  The default sampling experiment samples the program counters at regular intervals and records the total program time and the absolute and relative times each program counter was recorded. The default sampling interval is 10,000 microseconds by POSIX timer monotonic wall-clock time, but this can be changed using the \texttt{PAT_RT_INTERVAL} and \texttt{PAT_RT_INTERVAL_TIMER} environment variables.

- **samp_pc_ovfl**
  
  This experiment samples the program counters at the overflow of a specified hardware performance counter. The counter and overflow value are specified using the \texttt{PAT_RT_PERFCTR} environment variable. The default overflow counter is \texttt{cycles} and the default overflow frequency equates to an interval of 1,000 microseconds.
samp_cs_time

This experiment is similar to the samp_pc_time experiment, but samples the call stack at the specified interval and returns the total program time and the absolute and relative times each call stack counter was recorded.

samp_cs_ovfl

This experiment is similar to the samp_pc_ovfl experiment but samples the call stack.

samp_ru_time

(Deferred implementation) This experiment is similar to the samp_pc_time experiment but samples system resources.

samp_ru_ovfl

(Deferred implementation) This experiment is similar to the samp_pc_ovfl experiment but samples system resources.

samp_heap_time

(Deferred implementation) This experiment is similar to the samp_pc_time experiment but samples dynamic heap memory management statistics.

samp_heap_ovfl

(Deferred implementation) This experiment is similar to the samp_pc_time experiment but samples dynamic heap memory management statistics.
Using the CrayPat Run Time Environment

3.2.3.1 Trace-enhanced Sampling

Environment variable: PAT_RT_SAMPLING_MODE

If you use `pat_build` to instrument a program for a tracing experiment and then use `PAT_RT_EXPERIMENT` to specify a sampling experiment, trace-enhanced sampling is enabled and affects both user-defined functions and predefined function groups.

Trace-enhanced sampling is affected by the `PAT_RT_SAMPLING_MODE` environment variable. This variable can have one of the following values:

- **0**: Ignore trace-enhanced sampling. Perform a normal tracing experiment. (Default)
- **1**: Enable raw sampling. Any traced functions present in the instrumented program are ignored.
- **3**: Enable bubble sampling. Traced functions and any functions they call return a sample program counter address mapped to the trace function.

Trace-enhanced sampling is also affected by any other environment variables that affect a sampling experiment.

3.2.4 Improving Tracebacks

In normal operation, CrayPat does not write data files until either the buffer is full or the program reaches the end of planned execution. If your program aborts during execution and produces a core dump, performance analysis data is normally either lost or incomplete.
If this happens, consider setting `PAT_RT_SETUP_SIGNAL_HANDLERS` to 0, in order to bypass the CrayPat run time library and capture the signals the program receives. This results in an incomplete experiment file but a more accurate traceback, which may make it easier to determine why the program is aborting.

Alternatively, consider setting `PAT_RT_WRITE_BUFFER_SIZE` to a value smaller than the default value of 8MB, or using the `PAT_flush_buffer` API call to force CrayPat to write data. Both will cause CrayPat to write data more often, which results in a more-complete experiment data file.

### 3.2.5 Measuring MPI Load Imbalance

Environment variable: `PAT_RT_MPI_SYNC`

In MPI programs, time spent waiting at a barrier before entering a collective can be a significant indication of load imbalance. The `PAT_RT_MPI_SYNC` environment variable, if set, causes the trace wrapper for each collective subroutine to measure the time spent waiting at the barrier call before entering the collective. This time is reported by `pat_report` in the function group `MPI_SYNC`, which is separate from the `MPI` function group, which shows the time actually spent in the collective.

This environment variable affects tracing experiments only and is set on by default.

### 3.2.6 Monitoring Performance Counters

Environment variable: `PAT_RT_PERFCTR`

Use this environment variable to specify hardware, network, accelerator, power management, and AMD Interlagos Northbridge events to be monitored while performing tracing experiments.

Counter events are specified in a comma-separated list. Event names and groups from all three components may be mixed as needed; the tool is able to parse the list and determine which event names or group numbers apply to which components. To list the names of the individual CPU, GPU, network, power management, or AMD Northbridge events on your system, use the `papi_avail` and `papi_native_avail` man pages.

**Note:** Remember, to get useful information, `papi_avail` or `papi_native_avail` must be run on the compute nodes via the `aprun` command, not run from the login node or `esLogin` command line.

Depending on the counter selected, individual counter events are specified in one of three ways:

- use the performance counter event name, as given by `papi_avail` or `papi_native_avail`
• use the performance counter event name followed by the @ symbol and a value, to indicate a non-default overflow value used by the sampling-by-overflow experiments

• use the performance counter event name followed by the = sign and a value, to assign a value to a configuration event on an Aries network router

This environment variable also supports the use of keywords. The keywords currently recognized are:

• domain:u — specify that hardware counters are active in the user's domain

• domain:k — specify that hardware counters are active in the kernel (OS) domain

• domain:x — specify that hardware counters are active in the exception domain

• mpx — enable multiplexing for CPU events

The behavior of the PAT_RT_PERFCTR environment variable is also affected by the PAT_RT_PERFCTR_FILE and PAT_RT_PERFCTR_FILE_GROUP environment variables. These are described in detail in the intro_craypat(1) man page.

3.2.6.1 Hardware Counters

Alternatively, predefined counter group numbers can be used in addition to or in place of individual event names, to specify one or more predefined performance counter groups. The valid counter CPU group numbers are listed in the hwpc(5) man page.

3.2.6.2 Network Counters

Alternatively, predefined network counter group names can be used in addition to or in place of individual event names, to specify one or more predefined network counter groups. The valid predefined network counter names are listed in the nwpc(5) man page.

For more information about available network performance counters:

• On Gemini-based systems, either read the technical note Using the Cray Gemini Hardware Counters, read the files $CRAYPAT_ROOT/share/Counters.papi_gemini or $CRAYPAT_ROOT/share/Counters.papi_gemini.xml, or view the counters→gemini topics in pat_help.
• On Aries-based systems, either read the technical note *Using the Aries Hardware Counters*, read the files $CRAYPAT_ROOT/share/Counters.papi_aries or $CRAYPAT_ROOT/share/Counters.papi_aries.xml, or view the counters→aries topics in pat_help.

Network performance counter environment variables should be set only during tracing experiments. They are not useful for sampling experiments other than samp_pc_time.

### 3.2.6.3 Accelerator Counters

Alternatively, an `acgrp` value can be used in place of the list of event names, to specify a predefined performance counter accelerator group. The valid `acgrp` names are listed in the `accpc(5)` man page or on your system in $CRAYPAT_ROOT/share/CounterGroups.accelerator, where `accelerator` is the accelerator GPU used on your system.

**Note:** If the `acgrp` value specified is invalid or not defined, `acgrp` is treated as a counter event name. This can cause instrumented code to generate "invalid ACC performance counter event name" error messages or even abort during execution. Always verify that the `acgrp` values you specify are supported on the type of GPU accelerators that you are using.

Accelerated applications cannot be compiled with `-h profile_generate`, therefore GPU accelerator performance statistics and loop profile information cannot be collected simultaneously.

### 3.2.6.4 Power Management Counters

Cray XC30 and Cray XC30-AC systems support two types of power management counters. The PAPI Cray RAPL component provides socket-level access to Intel Running Average Power Limit (RAPL) measurement counters, while the similar PAPI Cray Power Management (PM) performance counters provide compute node-level access to additional power management counters. Together, these counters enable you to monitor and report energy usage during program execution.

**Note:** Base support is provided in CLE 5.0.UP03. Accelerator support requires CLE 5.1 or higher.

CrayPat supports experiments that make use of both sets of counters. These counters are accessed through use of the `PAT_RT_PERFCTR` set of run time environment variables. When RAPL counters are specified, one core per socket is tasked with collecting and recording the specified events. When PM performance counters are specified, one core per compute node is tasked with collecting and recording the specified events. The resulting metrics appear on text reports.
To list the available events, use the `PAPI_native_avail` command on a compute node and filter for the desired PAPI components. For example:

```
% aprun papi_native_avail -i crayrapl
% aprun papi_native_avail -i craypm
```

For more information about the RAPL and PM performance counters, see the `rapl(5)` and `pmpc(5)` man pages.
The *pat_report* command is the text reporting component of the Cray Performance Analysis Tools suite. After you use the *pat_build* command to instrument your program, set the run time environment variables as desired, and then execute your program, use the *pat_report* command to generate text reports from the resulting data and export the data for use in other applications.

The *pat_report* command is documented in detail in the *pat_report*(1) man page. Additional information can be found in the online help system under *pat_help report*.

### 4.1 Using Data Files

The data files generated by CrayPat vary depending on the type of program being analyzed, the type of experiment for which the program was instrumented, and the run time environment variables in effect at the time the program was executed. In general, the successful execution of an instrumented program produces one or more `.xf` files, which contain the data captured during program execution.

Unless specified otherwise using run time environment variables, these file names have the following format:

```
a.out+pat[+app#]+PID-node[s|t].xf
```

Where:

- **a.out**: The name of the instrumented executable.
- **app#**: The application number of a Multiple Program Multiple Data (MPMD) job, if the executable is an MPMD application.
- **PID**: The process ID assigned to the instrumented executable at run time.
- **node**: The physical node ID upon which the rank zero process was executed.
- **s|t**: The type of experiment performed, either `s` for sampling or `t` for tracing.
Use the `pat_report` command to process the information in individual `.xf` files or directories containing `.xf` files. Upon execution, `pat_report` automatically generates an `.ap2` file, which is both a self-contained archive that can be reopened later using the `pat_report` command and the exported-data file format used by Cray Apprentice2.

**Note:** If the executable was instrumented with the `pat_build -O apa` option, running `pat_report` on the `.xf` file(s) also produces an `.apa` file, which is the file used by Automatic Profiling Analysis. See Using Automatic Profiling Analysis on page 25.

### 4.2 Producing Reports

To generate a report, use the `pat_report` command to process your `.xf` file or directory containing `.xf` files.

```bash
> pat_report a.out+p+PID-node.t.xf
```

The complete syntax of the `pat_report` command is documented in the `pat_report(1)` man page.

**Note:** Running `pat_report` automatically generates an `.ap2` file, which is both a self-contained archive that can be reopened later using the `pat_report` command and the exported-data file format used by Cray Apprentice2. Also, if the executable was instrumented with the `pat_build -O apa` option, running `pat_report` on the `.xf` file(s) or the associated `.ap2` file produces an `.apa` file, which is the file used by Automatic Profiling Analysis. See Using Automatic Profiling Analysis on page 25.

The `pat_report` command is a powerful report generator with a wide range of user-configurable options. However, the reports that can be generated are first and foremost dependent on the kind and quantity of data captured during program execution. For example, if a report does not seem to show the level of detail you are seeking when viewed in Cray Apprentice2, consider rerunning your program with different `pat_build` options, or different or additional run time environment variable values. Note that setting `PAT_RT_SUMMARY` set to zero (disabled) will enable time-line panels in Cray Apprentice2, but will not affect the reports available from `pat_report`. 
4.2.1 Using Predefined Reports

The easiest way to use pat_report is by using an -O option to specify one of the predefined reports. For example, enter this command to see a top-down view of the calltree.

```bash
> pat_report -O calltree datafile.xf
```

**Note:** In many cases there is a dependency between the way in which a program is instrumented in pat_build and the data subsequently available for use by pat_report. For example, you must instrument the program using the pat_build -g heap option (or one of the equivalent user-defined pat_build options) in order to get useful data on the pat_report -O heap report, or use the pat_build -g mpi option (or one of the equivalent user-defined pat_build options) in order to get useful data on the pat_report -O mpi_callers report.

The predefined reports currently available can be listed with pat_report -O -h. They include:

- **accelerator**
  
  Show calltree of accelerator performance data sorted by host time.

- **accpc**
  
  Show accelerator performance counters.

- **acc_fu**
  
  Show accelerator performance data sorted by host time.

- **acc_time_fu**
  
  Show accelerator performance data sorted by accelerator time.

- **acc_time**
  
  Show calltree of accelerator performance data sorted by accelerator time.

- **acc_show_by_ct**
  
  (Deferred implementation) Show accelerator performance data sorted alphabetically.

- **affinity**
  
  Shows affinity bitmask for each node. Can use -s pe=ALL and -s th=ALL to see affinity for each process and thread, and use -s filter_input=expression to limit the number of PEs shown.

- **profile**
  
  Show data by function name only

- **callers** (or **ca**)
  
  Show function callers (bottom-up view)

- **calltree** (or **ct**)
  
  Show calltree (top-down view)

- **ca+src**
  
  Show line numbers in callers
ct+src  Show line numbers in calltree

heap  Implies heap_program, heap_hiwater, and heap_leaks. Instrumented programs must be built using the pat_build -g heap option in order to show heap_hiwater and heap_leaks information.

heap_program  

Compare heap usage at the start and end of the program, showing heap space used and free at the start, and unfreed space and fragmentation at the end.

heap_hiwater  

If the pat_build -g heap option was used to instrument the program, this report option shows the heap usage "high water" mark, the total number of allocations and frees, and the number and total size of objects allocated but not freed between the start and end of the program.

heap_leaks  If the pat_build -g heap option was used to instrument the program, this report option shows the largest unfreed objects by call site of allocation and PE number.

kern_stats  Show kernel-level statistics including average kernel grid size, average block size, and average amount of shared memory dynamically allocated for the kernel.

load_balance  

Implies load_balance_program, load_balance_group, and load_balance_function. Show PEs with maximum, minimum, and median times.

load_balance_program
load_balance_group
load_balance_function

For the whole program, groups, or functions, respectively, show the imb_time (difference between maximum and average time across PEs) in seconds and the imb_time% (imb_time/max_time * NumPEs/(NumPEs - 1)). For example, an imbalance of 100% for a function means that only one PE spent time in that function.

load_balance_cm  

If the pat_build -g mpi option was used to instrument the program, this report option shows the load balance by group with collective-message statistics.
load_balance_sm

If the pat_build -g mpi option was used to instrument the program, this report option shows the load balance by group with sent-message statistics.

load_imbalance_thread

Shows the active time (average over PEs) for each thread number.

loops

If the compiler -h profile_generate option was used when compiling and linking the program, display loop count and optimization guidance information.

mesh_xyz

Show the coordinates in the network mesh.

mpi_callers

Show MPI sent- and collective-message statistics

mpi_sm_callers

Show MPI sent-message statistics

mpi_coll_callers

Show MPI collective-message statistics

mpi_dest_bytes

Show MPI bin statistics as total bytes

mpi_dest_counts

Show MPI bin statistics as counts of messages

mpi_sm_rank_order

Calculate a suggested rank order based on MPI grid detection and MPI point-to-point message optimization. Uses sent-message data from tracing MPI functions to generate suggested MPI rank order information. Requires the program to be instrumented using the pat_build -g mpi option.

mpi_rank_order

Calculate a rank order to balance a shared resource such as USER time over all nodes. Uses time in user functions, or alternatively, any other metric specified by using the -s mro_metric options, to generate suggested MPI rank order information.

mpi_hy_rank_order

Calculate a rank order based on a hybrid combination of mpi_sm_rank_order and mpi_rank_order.
nids  Show PE to NID mapping.

nwpc  Program network counter activity.

profile_nwpc

NWPC data by Function Group and Function. Table shown by default if NWPC counters are present in the .ap2 file.

profile_pe.th

Show the imbalance over the set of all threads in the program.

profile_pe_th

Show the imbalance over PEs of maximum thread times.

profile_th_pe

For each thread, show the imbalance over PEs.

program_time

Shows which PEs took the maximum, median, and minimum time for the whole program.

read_stats
write_stats

If the pat_build -g io option was used to instrument the program, these options show the I/O statistics by filename and by PE, with maximum, median, and minimum I/O times.

samp_profile+src

Show sampled data by line number with each function.

thread_times

For each thread number, show the average of all PE times and the PEs with the minimum, maximum, and median times.

Note: By default, all reports show either no individual PE values or only the PEs having the maximum, median, and minimum values. The suffix _all can be appended to any of the above options to show the data for all PEs. For example, the option load_balance_all shows the load balance statistics for all PEs involved in program execution. Use this option with caution, as it can yield very large reports.
4.2.2 User-defined Reports

In addition to the -O predefined report options, the pat_report command supports a wide variety of user-configurable options that enable you to create and generate customized reports. These options are described in detail in the pat_report(1) man page and examples are provided in the pat_help online help system.

If you want to create customized reports, pay particular attention to the -s, -d, and -b options.

- **s**: These options define the presentation and appearance of the report, ranging from layout and labels, to formatting details, to setting thresholds that determine whether some data is considered significant enough to be worth displaying.

- **d**: These options determine which data appears on the report. The range of data items that can be included also depends on how the program was instrumented, and can include counters, traces, time calculations, mflop counts, heap, I/O, and MPI data. As well, these options enable you to determine how the values that are displayed are calculated.

- **b**: These options determine how data is aggregated and labeled in the report summary.

For more information, see the pat_report(1) man page. Additional information and examples can be found in the pat_help online help system.

4.3 Exporting Data

When you use the pat_report command to view an .xf file or a directory containing .xf files, pat_report automatically generates an .ap2 file, which is a self-contained archive file that can be reopened later using either pat_report or Cray Apprentice2. No further work is required in order to export data for use in Cray Apprentice2.

The pat_report -f option also enables you to export data to ASCII text or XML-format files. When used in this manner, pat_report functions as a data export tool. The entire data file is converted to the target format, and the pat_report filtering and formatting options are ignored.

4.4 Automatic Profiling Analysis

Assuming your executable was instrumented using the pat_build -O apa option (which is the default behavior), running pat_report on the .xf data file also produces an .apa file containing the recommended parameters for reinstrumenting the program for more detailed performance analysis. For more information about Automatic Profiling Analysis, see Using Automatic Profiling Analysis on page 25.
4.5 MPI Automatic Rank Order Analysis

By default MPI program ranks are placed on compute node cores sequentially, in SMP style, as described in the intro_mpi(3) man page. You can use the MPICH_RANK_REORDER_METHOD environment variable to override this default placement, and in some cases achieve significant improvements in performance by placing ranks on cores so as to optimize use of shared resources such as memory or network bandwidth.

The Cray Performance Analysis Tools suite provides several ways to help you optimize MPI rank ordering. If you already understand your program's patterns of communications well enough to specify an optimized rank order without further assistance, you can use the grid_order utility to generate a rank order list that can be used as an input to the MPICH_RANK_REORDER_METHOD environment variable. For more information, see the grid_order(1) man page.

Alternatively, to use CrayPat to perform automatic rank order analysis and generate recommended rank-order placement information, follow these steps.

Procedure 4. Using Automatic Rank Order Analysis

1. Instrument your program using either the pat_build -g mpi or -O apa option.
2. Execute your program.
3. Use the pat_report command to generate a report from the resulting .xf data files.

When certain conditions are met (job size, data availability, etc.), pat_report will attempt to detect a grid topology and evaluate alternative rank orders for opportunities to minimize off-node message traffic, while also trying to balance user time across the cores within a node. These rank-order observations appear on the resulting profile report, and depending on the results, pat_report may also automatically generate one or more MPICH_RANK_ORDER files for use with the MPICH_RANK_REORDER_METHOD environment variable in subsequent application runs.

4.5.1 Forcing Rank Order Analysis

To force pat_report to generate an MPICH_RANK_ORDER file, use one of these options.

- -O mpi_sm_rank_order
- -O mpi_rank_order
- -O mpi_hy_rank_order
4.5.1.1 -O mpi_sm_rank_order

The -O mpi_sm_rank_order option displays a rank-order table based on MPI sent-message data (message sizes or counts, and rank-distances). pat_report attempts to detect a grid topology and evaluates alternative rank orders that minimize off-node message traffic. This has a prerequisite that pat_report was invoked with either the -g mpi or -O apa option. If successful, a MPICH_RANK_ORDER_Grid file is generated which can be used to dictate the rank order of a subsequent job. Instructions for doing so are included in the file.

**Note:** The grid detection algorithm used in the sent-message rank-order report looks for, at most, patterns in three dimensions. Also, note that while use of an alternative rank order may improve performance of the targeted metric (i.e., MPI message delivery), the effect on the performance of the application as a whole is unpredictable.

A number of related -s options are available to tune the mpi_sm_rank_order report. These include:

```
mro_sm_metric=Dm|Dc
```

Used with the -O mpi_sm_rank_order option. If set to Dm, the metric is the sum of P2P message bytes sent and received. If set to Dc, the metric is the sum of P2P message counts sent and received.

Default: Dm

```
mro_mpi_pct=value
```

Specify the minimum percentage of total time that MPI routines must consume before pat_report will suggest an alternative rank order. This requires that the profile table be displayed in order to get the Total MPI Time.

Default: 10 (percent)

```
rank_cell_dim=m1xm2x...
```

Specify a set of cell dimensions to use for rank-order calculations. For example, -s rank_cell_dim=2x3.

```
rank_grid_dim=m1xm2x...
```

Specify a set of grid dimensions to use for rank-order calculations. For example, -s rank_grid_dim=8x5x3.

4.5.1.2 -O mpi_rank_order

The -O mpi_rank_order option generates an alternate rank-order based on a resource metric that can be compared across all PEs and balanced across all nodes. The default metric is USER Time, but other HWPC or derived metrics may be specified. If successful, this generates a MPICH_RANK_ORDER_USER_Time file.
The following related -s options are available to tune the mpi_rank_order report. These include:

\[ \text{mro\_metric=ti|...} \]

Any metric can be specified, but memory traffic hardware performance counter events are recommended.

Default: ti

\[ \text{mro\_group=USER|MPI|...} \]

If specified, the metric is computed only for functions in the specified group.

Default: USER

### 4.5.1.3 -O mpi_hy_rank_order

The -O mpi_hy_rank_order option generates a hybrid rank-order from the MPI sent-message and shared-resource metric algorithms in an attempt to gain improvements from both. This is only done for experiments that contain MPI sent-message statistics and whose jobs ran with at least 24 PEs per node. If successful, this generates a MPICH_RANK_ORDER.USER_Time_hybrid file.

This option supports the same -s options as both -O mpi_sm_rank_order and -O mpi_rank_order.

### 4.5.1.4 Observations and Suggestions

The following is an example showing the rank-order observations generated from default pat_report processing on data from a 2045 PE job running on 32 PEs/node. Additional explanations are found in lines beginning with the + character.

~~~~~~~~~~~ Observations and suggestions ~~~~~~~~~~~~

**MPI Grid Detection:**

There appears to be point-to-point MPI communication in a 35 X 60 grid pattern. The 20.3% of the total execution time spent in MPI functions might be reduced with a rank order that maximizes communication between ranks on the same node. The effect of several rank orders is estimated below.
A file named MPICH_RANK_ORDER.Grid was generated along with this report and contains usage instructions and the Custom rank order from the following table.

+ Note that the instructions for using each MPICH_RANK_ORDER file are included within that file.

+ This shows that the Custom rank order was able to arrange the ranks such that 34% of the total MPI message bytes sent per PE stayed within each local compute node (the higher the percentage the better). In this case, the Custom order was a little better than the default SMP order.

Metric-Based Rank Order:

When the use of a shared resource like memory bandwidth is unbalanced across nodes, total execution time may be reduced with a rank order that improves the balance. The metric used here for resource usage is: USER Time

+ USER Time is the default, but can be changed via the -s mro_metric option.

For each node, the metric values for the ranks on that node are summed. The maximum and average value of those sums are shown below for both the current rank order and a custom rank order that seeks to reduce the maximum value.

A file named MPICH_RANK_ORDER.USER_Time was generated along with this report and contains usage instructions and the Custom rank order from the following table.

+ The Node Metric Imbalance column indicates the difference between the maximum and average metric values over the set of compute nodes. A lower the imbalance value is better, as the maximum value is brought down closer to the average.
Hybrid Metric-Based Rank Order:

A hybrid rank order has been calculated that attempts to take both the MPI communication and USER Time resources into account. The table below shows the metric-based calculations along with the final on-node bytes/PE value. A MPICH_RANK_ORDER.USER_Time_hybrid file was generated along with this report and contains usage instructions for this custom rank order.

<table>
<thead>
<tr>
<th>Rank Order</th>
<th>Node Reduction</th>
<th>Maximum Value</th>
<th>Average Value</th>
<th>On-Node Bytes/PE%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current</td>
<td>8.95%</td>
<td>6.971e+04</td>
<td>6.347e+04</td>
<td>23.82%</td>
</tr>
<tr>
<td>Custom</td>
<td>2.70%</td>
<td>6.43%</td>
<td>6.523e+04</td>
<td>6.347e+04</td>
</tr>
</tbody>
</table>

It will usually be the case that the hybrid node imbalance and the on-node bytes/PE values are not quite as good as the best values in the MPI grid-based and the metric-based tables, but the goal is to get them as close as possible while gaining benefits from both methodologies.
Cray Apprentice2 is an interactive X Window System tool for visualizing and manipulating performance analysis data captured during program execution.

The number and appearance of the reports that can be generated using Cray Apprentice2 is determined solely by the kind and quantity of data captured during program execution. For example, setting the PAT_RT_SUMMARY environment variable to 0 (zero) before executing the instrumented program nearly doubles the number of reports available when analyzing the resulting data in Cray Apprentice2. However, it does so at the cost of much larger data files.

5.1 Launching the Program

To begin using Cray Apprentice2, load the perftools module. If this module is not part of your default work environment, type the following command to load it:

```
> module load perftools
```

To launch the Cray Apprentice2 application, enter this command:

```
> app2 &
```

Alternatively, you can specify the file name to open on launch:

```
> app2 myfile.ap2 &
```

Note: Cray Apprentice2 requires that your workstation be configured to host X Window System sessions. If the app2 command returns an "cannot open display" error, see your system administrator for information about configuring X Window System hosting.

The app2 command supports two options: --limit and --limit_per_pe. These options enable you to restrict the amount of data being read in from the data file. Both options recognize the K, M, and G abbreviations for kilo, mega, and giga; for example, to open an .ap2 data file and limit Cray Apprentice2 to reading in the first 3 million data items, type this command:

```
> app2 --limit 3M data_file.ap2 &
```
The \texttt{--limit} option sets a global limit on data size. The \texttt{--limit\_per\_pe} sets the limit on a per processing element basis. Depending on the nature of the program being examined and the internal structure of the data file being analyzed, the \texttt{--limit\_per\_pe} is generally preferable, as it preserves data parallelism.

\textbf{Note:} The \texttt{--limit} and \texttt{--limit\_per\_pe} options affect only .ap2 format data files created with versions of \texttt{pat\_report} prior to release 5.2.0. These options are ignored when opening data files created using \texttt{pat\_report} release 5.2.0 or later and will be removed in a future release.

For more information about the \texttt{app2} command, see the \texttt{app2(1)} man page.

\section{5.2 Opening Data Files}

If you specified a valid data file or directory on the \texttt{app2} command line, the file or directory is opened and the data is read in and displayed.

If you did not specify a data file or directory on the command line, the File Selection Window is displayed and you are prompted to select a data file or directory to open.

\textbf{Note:} The exact appearance of the File Selection window varies depending on which version of the Gimp Tool Kit (GTK) is installed on your X Windows System workstation.

After you select a data file, the data is read in. When Cray Apprentice2 finishes reading in the data, the Overview report is displayed.

\section{5.3 Basic Navigation}

Cray Apprentice2 displays a wide variety of reports, depending on the program being studied, the type of experiment performed, and the data captured during program execution. While the number and content of reports varies, all reports share the following general navigation features.

\begin{itemize}
  \item The \textbf{File menu} enables you to open data files or directories, capture the current screen display to a .jpg file, or exit from Cray Apprentice2.
  \item The \textbf{Data tab} shows the name of the data file currently displayed. You can have multiple data files open simultaneously for side-by-side comparisons of data from different program runs. Click a \textbf{data tab} to bring a data set to the foreground. Right-click the tab for additional options.
  \item The \textbf{Report toolbar} show the reports that can be displayed for the data currently selected. Hover the cursor over an individual report icon to display the report name. To view a report, click the icon.
  \item The \textbf{Report tabs} show the reports that have been displayed thus far for the data currently selected. Click a tab to bring a report to the foreground. Right-click a tab for additional report-specific options.
\end{itemize}
• The main display varies depending on the report selected and can be resized to suit your needs. However, most reports feature **pop-up tips** that appear when you allow the cursor to hover over an item, and **active data elements** that display additional information in response to left or right clicks.

• On many reports, the total duration of the experiment is shown as a graduated bar at the bottom of the report window. Move the **caliper points** left or right to restrict or expand the span of time represented by the report. This is a global setting for each data file: moving the caliper points in one report affects all other reports based on the same data, unless those other reports have been detached or frozen.

Most report tabs feature **right-click menus**, which display both common options and additional report-specific options. The common right-click menu options are described in **Table 2**. Report-specific options are described in **Viewing Reports on page 71**.

**Table 2. Common Panel Actions**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screendump</td>
<td>Capture the report or graphic image currently displayed and save it to a .jpg file.</td>
</tr>
<tr>
<td>Detach Panel</td>
<td>Display the report in a new window.</td>
</tr>
<tr>
<td>Remove Panel</td>
<td>Close the window and remove the report tab from the main display.</td>
</tr>
<tr>
<td>Panel Help</td>
<td>Display report-specific help, if available.</td>
</tr>
</tbody>
</table>

### 5.4 Viewing Reports

The reports Cray Apprentice2 produces vary depending on the types of performance analysis experiments conducted and the data captured during program execution. The report icons indicate which reports are available for the data file currently selected. Not all reports are available for all data.

The following sections describe the individual reports.

#### 5.4.1 Overview Report

The Overview Report is the default report. Whenever you open a data file, this is the first report displayed.
The Overview Report provides a high-level view of the program's performance characteristics, and is divided into five main areas of concern. These are:

- **Profile**: The center of the Overview window displays a bar graph designed to give you a high-level assessment of how much CPU time (as a percentage of wall-clock time) the program spent doing actual computation, versus Programming Model overhead (i.e., MPI communication, UPC or SHMEM data movement, OpenMP parallel region work, etc.) and I/O.

If the program uses GPUs, a second bar graph is displayed showing GPU time relative to wall-clock time. The numbers in the GPU bar graph are the percentages of total time that were spent in the specified GPU functions, and thus are not expected to equal 100% of the wall-clock time.

- **Function/Region Profile**: The Function/Region Profile in the upper-left corner of the Overview Report highlights the top time-consuming functions or regions in the code. Click on the pie chart to jump to the Profile Report.

- **Load Imbalance**: The Load Imbalance summary in the lower-left corner of the Overview Report highlights load imbalance, if detected, as a percentage of wall-clock time. Click on the scales to jump to the Call Tree Report.

If an "i" ("information") icon is displayed, you can hover the cursor over it to see additional grid detection information and rank placement suggestions.

- **Memory Utilization**: The Memory Utilization summary in the upper-right corner of the Overview Report highlights poor memory hierarchy utilization, if detected, including TLB and cache utilization.

If an "i" ("information") icon is displayed, you can hover the cursor over it to see additional observations.

- **Data Movement**: The Data Movement summary in the lower-right corner of the Overview Report identifies data movement bottlenecks, if detected.
5.4.2 Profile Report

The Profile Report is a good general display showing where your program spent the most time, a good indicator of how much time your program is spending performing which activities, and a good place to start looking for load imbalance. Depending on the data collected, this report initially displays as one or more pie charts. When the Profile Report is displayed, look for:

- In the pie chart on the left, the calls, functions, regions, and loops in the program, sorted by the number of times they were invoked and expressed as a percentage of the total call volume.
- In the pie chart on the right, the calls, functions, regions, and loops, in the program, sorted by the amount of time spent performing the calls or functions and expressed as a percentage of the total program execution time.
- Hover the cursor over any section of a pie chart to display a pop-up window containing specific detail about that call, function, region, or loop.
- Right-click on any call or function on a pie chart to display the "Fastbreak" option. Click Fastbreak to jump directly to this call or function in the Call Tree graph.
- Right-click the Report Tab to display a pop-up menu that lets you show or hide compute time. Hiding compute time is useful if you want to focus on the communications aspects of the program.

To explore this further, click any function of interest to display a Load Balance Report for that function.

The Load Balance Report shows:

- The load balance information for the function you selected on the Profile Report. This report can be sorted by either PE, Calls, or Time. Click a column heading to sort the report by the values in the selected column.
- The minimum, maximum, and average times spent in this function, as well as standard deviation.
- Hover the cursor over any bar to display PE-specific quantitative detail.

Alternately, click the Toggle (the double-headed arrow in the upper right corner of the report tab) to view the Profile Report as a bar graph, or click the Toggle again to view the Profile Report as a text report. In both bar graph and text report modes, the Load Balance and "Fastbreak" functions are available by clicking or right-clicking on a call or function.
The text version of the Profile Report is a table showing the time spent by function, as both a wall clock time and percentage of total run time. This report also shows the number of calls to the function, the number of call sites in the code that call the function, the extent to which the call is imbalanced, and the potential savings that would result if the function were perfectly balanced.

This is an active report. Click on any column heading to sort the report by that column, in ascending or descending order. In addition, if a source file is listed for a given function, you can click on the function name and open the source file at the point of the call.

Look for routines with high usage, a small number of call sites, and the largest imbalance and potential savings, as these are the often the best places to focus your optimization efforts.

Together, the Profile and Load Balance reports provide a good look at the behavior of the program during execution and can help you identify opportunities for improving code performance. Look for functions that take a disproportionate amount of total execution time and for PEs that spend considerably more time in a function than other PEs do in the same function. This may indicate a coding error, or it may be the result of a data-based load imbalance.

To further examine load balancing issues, examine the Mosaic report (if available), and look for any communication "hotspots" that involve the PEs identified on the Load Balance Report.

### 5.4.3 Text Report

The Text Report option enables you to access pat_report text reports through the Cray Apprentice2 user interface and to generate new text reports with the click of a button.

### 5.4.4 Environment Reports

The Environment Reports provide general information about the conditions under which the data file currently being examined was created. As a rule, this information is useful only when trying to determine whether changes in system configuration have affected program performance.

The Environment Reports consists of four panes. The Env Vars pane lists the values of the system environmental variables that were set at the time the program was executed.

**Note:** This does not include the pat_build or CrayPat environment variables that were set at the time of program execution.

The System Info pane lists information about the operating system.
The **Resource Limits** pane lists the system resource limits that were in effect at the time the program was executed.

The **Heap Info** pane lists heap usage information.

There are no active data elements or right-click menu options in any of the Environment Reports.

### 5.4.5 Traffic Report

The Traffic Report shows internal PE-to-PE traffic over time. The information on this report is broken out by communication type (read, write, barrier, and so on). While this report is displayed, you can:

- Hover over an item to display quantitative information.
- Zoom in and out, either by using the zoom buttons or by drawing a box around the area of interest.
- Right-click an area of interest to open a popup menu, which enables you to hide the origin or destination of the call or go to the call site in the source code, if the source file is available.
- Right-click the report tab to access alternate zoom in and out controls, or to filter the communications shown on the report by the duration of the messages.

Filtering messages by duration is useful if you are only interested in a particular group of messages. For example, to see only the messages that take the most time, move the filter caliper points to define the range you want, and then click the **Apply** button.

The Traffic Report is often quite dense and typically requires zooming in to reveal meaningful data. Look for large blocks of barriers that are being held up by a single PE. This may indicate that the single PE is waiting for a transfer, or it can also indicate that the rest of the PEs are waiting for that PE to finish a computational piece before continuing.

### 5.4.6 Mosaic Report

The Mosaic Report depicts the matrix of communications between source and destination PEs, using colored blocks to represent the relative communication times between PEs. By default, this report is based on average communication times. Right-click on the report tab to display a popup menu that gives you the choice of basing this report on the Total Calls, Total Time, Average Time, or Maximum Time.

The graph is color-coded. Light green blocks indicates good values, while dark red blocks may indicate problem areas. Hover the cursor over any block to show the actual values associated with that block.
Use the diagonal scrolling buttons in the lower right corner to scroll through the report and look for red "hot spots." These are generally an indication of bad data locality and may represent an opportunity to improve performance by better memory or cache management.

5.4.7 Activity Report

The Activity Report shows communication activity over time, bucketed by logical function such as synchronization. Compute time is not shown.

Look for high levels of usage from one of the function groups, either over the entire duration of the program or during a short span of time that affects other parts of the code. You can use the calipers to filter out the startup and closeout time, or to narrow the data being studied down to a single iteration.

5.4.8 Call Tree

The Call Tree shows the calling structure of the program as it ran and charts the relationship between callers and callees in the program. This report is a good way to get a sense of what is calling what in your program, and how much relative time is being spent where.

Each call site is a separate node on the chart. The relative horizontal size of a node indicates the cumulative time spent in the node's children. The relative vertical size of a node indicates the amount of time being spent performing the computation function in that particular node.

Nodes that contain only callers are green in color. Nodes for which there is performance data are dark green, while light-green nodes have no data of their own, only inclusive data bubbled up from their progeny.

By default, routines that do not lead to the top routines are hidden.

Nodes that contain callees and represent significant computation time also include stacked bar graphs, which present load-balancing information. The yellow bar in the background shows the maximum time, the pale purple in the foreground shows the minimum time, and the purple bar shows the average time spent in the function. The larger the yellow area visible within a node, the greater the load imbalance.
While the Call Tree report is displayed, you can:

- Hover the cursor over any node to further display quantitative data for that node.
- Double-click on leaf node to display a Load Balance report for that call site.
- If a "?" (question mark) icon is displayed on any node, this indicates that significant additional information pertinent to this node is available: for example, that the node has the highest load-imbalance time in the program and thus is a good candidate for optimization. Hover the cursor over the "?" icon to display additional information.
- Right-click the report tab to display a popup menu. The options on this menu enable you to change this report so that it shows all times as percentages or actual times, or highlights imbalance percentages and the potential savings from correcting load imbalances. This menu also enables you to filter the report by time, so that only the nodes representing large amounts of time are displayed, or to unhide everything that has been hidden by other options and restore the default display.
- Right-click any node to display another popup menu. The options on this menu enable you to hide this node, use this node as the base node (thus hiding all other nodes except this node and its children), jump to this node's caller, or go to the source code, if available.
- Use the zoom control in the lower right corner to change the scale of the graph. This can be useful when you are trying to visualize the overall structure.
- Use the Search control in the lower center to search for a particular node by function name.
- Use the \( > > \) toggle in the lower left corner to show or hide an index that lists the functions on the graph by name. When the index is displayed, you can double-click a function name in the index to find that function in the Call Tree.

### 5.4.9 I/O Rates

The I/O Rates Report is a table listing quantitative information about the program's I/O usage. The report can be sorted by any column, in either ascending or descending order. Click on a column heading to change the way that the report is sorted.

Look for I/O activities that have low average rates and high data volumes. This may be an indicator that the file should be moved to a different file system.

**Note:** This report is available only if I/O data was collected during program execution. See Chapter 2, Using `pat_build` on page 25 and the `pat_build(1)` man page for more information.
5.4.10 Hardware Reports

The Hardware reports are available only if hardware counter information has been captured. There are two Hardware reports:

- Hardware Counters Overview
- Hardware Counters Plot

5.4.10.1 Hardware Counters Overview Report

The Hardware Counters Overview Report is a bar graph showing hardware counter activity by call and function, for both actual and derived PAPI metrics. While this report is displayed, you can:

- Hover the cursor over a call or function to display quantitative detail.
- Click the "arrowhead" toggles to show or hide more information.

5.4.10.2 Hardware Counters Plot

The Hardware Counters Plot displays hardware counter activity over time as a trend plot. Use this report to look for correlations between different kinds of activity. This report is most useful when you are more interested in knowing when a change in activity happened rather than in knowing the precise quantity of the change.

Look for slopes, trends, and drastic changes across multiple counters. For example, a sudden decrease in floating point operations accompanied by a sudden increase in L1 cache activity may indicate a problem with caching or data locality. To zero-in on problem areas, use the calipers to narrow the focus to time-spans of interest on this graph, and then look at other reports to learn what is happening at these times.

To display the value of a specific data point, along with the maximum value, hover the cursor over the area of interest on the chart.

5.4.11 GPU Time Line

The GPU Time Line shows concurrent activity on the CPU (host) and GPU (accelerator). This helps you visualize if and how CPU and GPU events overlap in time.

Note: This report is available only with a full trace data file.
5.4.11.1 CPU Call Stack and GPU Stream

The GPU Time Line report is divided into two general areas. The upper half of the window shows CPU Stack and GPU Stream data over time, and contains the related controls.

Stack

The **Stack** display shows the call stack levels of the program running on the CPU, starting with 0 (main) at the top. Use the scroll bar controls at the right end of the display to move through the call stack levels. If the window is resized so that all levels are visible, the scroll bar controls are inactive.

Each box in the **Stack** display represents an interval of execution, typically an instance of a function call. Vertically stacked boxes represent functions calling other functions. Green boxes are transfers between the CPU and GPU, red boxes are wait-related functions, and all others are blue. You can hover the cursor over a box to see a popup that describes the event; when you do so, any related GPU events also change color to highlight the relationship. Alternatively, you can left-click while hovering to turn the popup into a separate window.

Stream

The **Stream** displays shows the related GPU activity during the same period in time. Each box represents GPU stream activity, and you can use the scroll bar controls at the right end of the **Stream** display to move through the stream activity levels. The color-coding and hover/popup behavior are the same as for the **Stack** section.

You can use the windowshade control between the **Stack** and **Stream** displays to change the relative sizes of the displays.

time line

The horizontal scroll bar immediately below the **Stream** display shows the time interval represented by the **Stack** and **Stream** displays. The red vertical bar is the center-point of the current display, and also the value shown in the **Time** entry field. You can left-click anywhere on the scroll bar to re-center the display on another point in time. The scale of this scroll bar is determined by the **Magnify** control.

Magnify

The **Magnify** control determines the scale of the horizontal time line that in turn defines the region shown in the **Stack** and **Stream** displays. At a setting of 1.0, the time line duration is the same as the entire duration of the program. Use higher magnification levels to reveal finer granularity in the data.

Time

Alternatively, you can enter a value in the **Time** entry field, jump directly to that point in time during program execution, and re-center the display on that point in time.
Func

The **Func** entry field enables you to search for instances of a specific function by name. As you begin to enter a function name, a drop-down list appears, showing all matching names within the program. After you have entered or selected the function you want to find, use the **Prev** and **Next** buttons to move to the previous or next instance of the function.

### 5.4.11.2 Histogram

The lower portion of the GPU Time Line report window shows the histogram and related controls.

**Kern, In, Out, Wait**

The radio buttons to the left of the histogram select what information is displayed in the histogram and are mutually exclusive. **Wait** displays an aggregate of CPU functions that wait for something else to complete. **Kern, In, and Out** are GPU-related and quantify where time on the GPU is spent. Each histogram shows how much time was spent in the selected category during the specified interval. The measurements are in percent, with the full height of the window representing 100%.

**PE, TH**

The **PE** and **TH** entry boxes enable you to filter what is displayed by PE and thread. These are set to zero by default.

**time line, Zoom**

By default, the time line below the histogram represents the entire duration of the program execution, unless you use the **Zoom** control to change the scale. The red vertical bar is the center-point of the current display. The horizontal scroll bar below the histogram is generally inactive, unless you use the **Zoom** control to zoom-in on some activity.
Reveal is Cray's next-generation integrated performance analysis and code optimization tool. Reveal extend's Cray's existing performance measurement, analysis, and visualization technology by combining run time performance statistics and program source code visualization with Cray Compiling Environment (CCE) compile-time optimization feedback.

Reveal supports source code navigation using whole-program analysis data provided by the Cray Compiling Environment, coupled with performance data collected during program execution by the Cray performance tools, to understand which high-level serial loops could benefit from improved parallelism. Reveal provides enhanced loopmark listing functionality, dependency information for targeted loops, and assists users optimizing code by providing variable scoping feedback and suggested compiler directives.

Note: Reveal works with the Cray Compiling Environment (CCE) only. It does not work with other third-party compilers at this time.

### 6.1 Launching the Program

To begin using Reveal, load the `perftools` module, and then enter the `reveal` command:

```
> module load perftools
> reveal
```

If no files are specified on the command line, the user can open an existing program library file by selecting the File→Open option.

To launch Reveal and open a specific `program_library` file:

```
> reveal my_program_library.pl
```

Note: The .pl file name extension is not required. It is added in these examples to help improve clarity.

To launch Reveal and specify both a `program_library` and a `performance_data` file:

```
> reveal my_program_library.pl my_program.ap2
```

Reveal includes an integrated help system. All other information about using Reveal is presented in the help system, which is accessible whenever Reveal is running by selecting Help from the menu bar.
Reveal is a GUI tool that requires that your workstation support the X Window System. Depending on your system configuration, you may need to use the `ssh -X` option to enable X Window System support in your shell session. Depending on your workstation configuration, you may also need to enable X Window System hosting on your workstation or load an X Window client such as Xming.

### 6.2 Using Reveal

Reveal can be used to open and explore a performance analysis data file, a program library file, or both at the same time. You can also use Reveal to open one type of data file for a program, and then open the other type later, to begin searching for correlations between code performance and optimizations. In general, though, the most common way to use Reveal consist of three steps: capturing run time performance data to generate loop work estimates, generating a program library file to capture and analyze compiler optimizations, and then integrating the two sets of data.

#### 6.2.1 Generating Loop Work Estimates

Loop work estimates are generated by compiling and linking with the CCE `-h profile_generate` option, and then using CrayPat to instrument the program for tracing, run the instrumented executable, and collect loop statistics. To generate a loop work estimate, follow these steps.

1. Make sure the following modules are loaded.
   ```bash
   > module load PrgEnv-cray
   > module load perftools
   ```

2. Compile and link the program with `-h profile_generate`.
   ```bash
   > ftn -c -h profile_generate my_program.f
   > ftn -o my_program -h profile_generate my_program.o
   ```

   **Note:** This option disables most automatic compiler optimizations, which is why Cray recommends generating this data separately from generating the `program_library` file. The `program_library` is most useful when generated from fully optimized code.

3. Instrument the program for tracing.
   ```bash
   > pat_build -w my_program
   ```

   This generates a new binary named `my_program+pat`.

4. Execute the instrumented program.
   ```bash
   > aprun -n pes ./my_program+pat
   ```

   This generates one or more raw data files in `.xf` format.
5. Process the raw data file for use by Reveal.

```
pat_report -o my_program.ap2 my_programXXX.xf > loops_report
```

This generates a performance data file (`my_program.ap2`) and a text report (`loops_report`).

### 6.2.2 Generating a Program Library

To generate a `program_library.pl` file, make sure the Cray (CCE) programming environment module is loaded, and then use the CCE `–h pl` option to generate the `program_library` in your current working directory.

```
module load PrgEnv-cray

> ftn -O3 -hpl=my_program.pl -c my_program_file1.f90
> ftn -O3 -hpl=my_program.pl -c my_program_file2.f90
> ftn -O3 -hpl=my_program.pl -c my_program_file3.f90
> ...
```

**Note:** The `–h` profile `generate` option disables most automatic compiler optimizations, which is why Cray recommends generating the `program_library` file separately from the loop work estimate. The `program_library` is most useful when generated from fully optimized code.

**Note:** You must keep your program library with your program source. Moving just the `program_library` file to another location and then opening it with Reveal is not supported.

### 6.2.3 Exploring the Results

After you have collected performance data from program execution and generated a `program_library` file, launch Reveal and use it to integrate the results and explore opportunities for code optimization.

```
> reveal my_program.pl my_program.ap2
```

Alternatively, you can launch Reveal with either the program library or performance data file, and then use the **File→Open** option to integrate the two data sets.

### 6.3 For More Information

Reveal is a new and evolving product, therefore documentation and training for this product are still in development. Reveal includes an integrated help system: further information about using Reveal is presented in the help system, which is accessible whenever Reveal is running by selecting **Help** from the menu bar.
Environment variables are used extensively to control the behavior of CrayPat and the collection and processing of experiment data. This appendix replicates the environment variable information found in the intro_craypat(1), pat_build(1), and pat_report(1) man pages. In the event of differences between this appendix and the man pages, the man pages are assumed to be more current.

A.1 pat_build Environment Variables

The following environment variables affect the operation of pat_build.

**PAT_BUILD_CLEANUP**

By default or if set to a nonzero value, the intermediate directory is removed. Set to zero to retain the directory after pat_build completes.

**PAT_BUILD_EMBED_RTENV**

Specifies one or more comma-separated CrayPat run time environment variables to embed in the instrumented program. The CrayPat run time environment variables must be set at the time the instrumented program is created. By default, all CrayPat run time environment variables that are set at the time the instrumented program is created are embedded in the instrumented program. If PAT_BUILD_EMBED_RTENV is set to zero, no CrayPat run time environment variables are embedded. For more information, see the description of the rtenv build directive, in Advanced Users: Environment Variables and Build Directives on page 31.

**PAT_BUILD_OPTIONS**

Specifies the pat_build options that are evaluated before any options on the command line.
PAT_BUILD_PAPI_DIR

Specifies the location of the PAPI run time library. If this environment variable is set, the directory path is valid, and the libpapi.a or libpapi.so file exists in the specified directory, the respective file is used to satisfy the PAPI requirements of the CrayPat run time library when the instrumented program is created.

Default: /opt/cray/papi/current_version

PAT_BUILD_PRMGT

If set to nonzero, forces Process Management (PMI) entry points to be loaded into the instrumented program. If set to zero, no additional PMI entry points are loaded into the instrumented program. If not set (default), PMI entry points are loaded into the instrumented program only if the programming models present in the input program dictate the need for PMI.

PAT_BUILD_VERBOSE

Specifies the detail level of the progress messages related to the instrumentation process. This value corresponds to the number of -v options specified.

PAT_LD_OBJECT_TMPDIR

Allows the user to change the location of the directory where CrayPat copies of object files that are in a /tmp directory. When set, pat_build writes copies of object files into the ${PAT_LD_OBJECT_TMPDIR}/.craypat/program-name/PID-of-link directory. The default value for PAT_LD_OBJECT_TMPDIR is $HOME.

To disable copying of object files, set this environment variable to 0 (zero).

A.2 Run Time Environment Variables

The run time environment variables communicate directly with an executing instrumented program and affect how data is collected and saved.

PAT_RT_ACC_ACTIVITY_BUFFER_SIZE

Specifies the size in bytes of the buffer used to collect records for the accelerator time line view in Cray Apprentice2. Size is not case-sensitive and can be specified in kilobytes (KB), megabytes (MB), or gigabytes (GB).

Default: 1MB
PAT_RT_ACC_RECORD

Overrides the programming model for which accelerator performance data is collected. The valid values are:

- **off**: Disables collection of accelerator performance data.
- **cce**: Collect performance data for applications compiled with CCE and using OpenACC directives.
- **cuda**: Collect performance data for CUDA applications.
- **pgi**: Collect performance data for applications using PGI accelerator directives.

Default: unset

PAT_RT_ACC_FORCE_SYNC

Forces accelerator synchronization in order to enable collection of accelerator time for asynchronous events.

Default: not enabled

PAT_RT_BUILD_ENV

Indicates if any runtime environment variables embedded in the instrumented program using the `pat_build rtenv` directive are ignored. If set to 0, all embedded environment variables in the instrumented program are ignored. If set to 1, all embedded environment variables are used.

Default: 1

A comma-separated list of environment variable names may follow 0 or 1. This is an exception list. If a list appears after 0, all embedded environment variables are ignored (unset) except the ones listed. Conversely, if a list appears after 1, all embedded environment variable are used except the ones listed.

If an environment variable that is embedded in the instrumented program is also set in the execution environment, the value set in the execution environment takes precedence and the value embedded in the instrumented program is ignored.

For more information about the `rtenv` directive, see Advanced Users: Environment Variables and Build Directives on page 31.
PAT_RT_CALLSTACK

Specifies the depth to which to trace the call stack for a given entry point when sampling or tracing. For example, if set to 1, only the caller of the entry point is recorded.

Default: 100 or to the main entry point, whichever is less

PAT_RT_CALLSTACK_BUFFER_SIZE

Specifies the size in bytes per thread of the runtime summary buffer used to collect function call stacks. Size is not case-sensitive and can be specified in kilobytes (KB), megabytes (MB), or gigabytes (GB).

Default: 4MB

PAT_RT_CHECKPOINT

Enable/disable checkpoint/restart on the executing instrumented program, for those systems that support Berkeley Lab Checkpoint Restart (BLCR). If set to 0, the executing instrumented program ignores the `cr_checkpoint` and `cr_restart` commands. A value greater than zero indicates the maximum number of checkpoint states collected. Checkpoints received by the instrumented program after this number is reached are ignored.

The instrumented program supports checkpoint/restart only if the original program supported checkpoint/restart.

Default: 32 (checkpoint/restart enabled, a maximum of 32 checkpoint states collected).

PAT_RT_COMMENT

Specifies an arbitrary string that is inserted into the experiment data file. The string is included in the report analysis done by `pat_report`.

Default: unset

PAT_RT_CONFIG_FILE

Specifies one or more configuration files that contain environment variables. Multiple file names are separated with the comma (,) character. Lines in the file that begin with the # character are interpreted as comments and ignored. If the file name specified begins with a question mark (?) character, and the file does not exist or is otherwise inaccessible, no fatal error is generated.
Environment variables are of the form defined by sh and ksh:
\[\text{name}=\text{value}.\]

After all files specified by the \texttt{PAT\_RT\_CONFIG\_FILE}
environment variable are processed, if the file
\texttt{$HOME/.craypatrc} exists, its contents are processed.
Next, if the file \texttt{./craypatrc} exists, its contents are processed.

The environment variables appear in the file(s) one per line.
Each subsequent environment variable name replaces the
value of the previous one with the same name. Typically, the
\texttt{PAT\_RT\_CONFIG\_FILE} environment variable is used by site
administrators to define default system-wide CrayPat runtime
environment variables. Users should exercise caution when changing
\texttt{PAT\_RT\_CONFIG\_FILE} or adding additional configuration files
to it.

Default: unset

\texttt{PAT\_RT\_EXIT\_AFTER\_INIT}

If nonzero, terminate execution after the initialization of the CrayPat runtime library is complete.

Default: 0

\texttt{PAT\_RT\_EXPERIMENT}

Identifies the experiment to perform.

By default, CrayPat instruments programs for Automatic Profiling Analysis. However, if a program is instrumented for a sampling experiment by using the \texttt{pat\_build -S} option, or for tracing by using the \texttt{pat\_build -w, -U, -u, -T, -t, or -g} options, then you can use the \texttt{PAT\_RT\_EXPERIMENT} environment variable to further specify the type of experiment to be performed.

If a program is instrumented for tracing and you then use \texttt{PAT\_RT\_EXPERIMENT} to specify a sampling experiment, trace-enhanced sampling is performed, subject to the rules established by the \texttt{PAT\_RT\_SAMPLING\_MODE} environment variable setting.

Depending on the options you select, it is possible to generate extremely large data files.
The valid experiments are:

```
samp_pc_time
```
Samples the program counter at a given time interval. This returns the total program time and the absolute and relative times each program counter was recorded. The default interval is 10,000 microseconds.

The default POSIX interval timer measures monotonic wall-clock time. This is changed using the `PAT_RT_INTERVAL_TIMER` runtime environment variable.

```
samp_pc_ovfl
```
Samples the program counter at a given overflow of a hardware performance counter. The hardware counter and its overflow value are separated by the `@` symbol and specified in a comma-separated list in the runtime variable `PAT_RT_PERFCTR`, i.e., `event-name@overflow-value`. The default overflow counter is cycles and the default overflow frequency equates to an interval of 1,000 microseconds.

```
samp_cs_time
```
Samples the call stack at a given time interval. This returns the total program time and the absolute and relative times each call stack counter was recorded, and is otherwise identical to the `samp_pc_time` experiment.

```
samp_cs_ovfl
```
Samples the call stack at a given overflow of a hardware performance counter. This experiment is otherwise identical to the `samp_pc_ovfl` experiment.

```
samp_ru_time
```
(Deferred implementation) Samples system resources at a given time interval. This experiment is otherwise identical to the `samp_pc_time` experiment.
samp_ru_ovfl

(Deferred implementation) Samples system resources at a given overflow of a hardware performance counter. This experiment is otherwise identical to the samp_pc_ovfl experiment.

samp_heap_time

(Deferred implementation) Samples dynamic heap memory management statistics at a given time interval. This experiment is otherwise identical to the samp_pc_time experiment.

samp_heap_ovfl

(Deferred implementation) Samples dynamic heap memory management statistics at a given overflow of a hardware performance counter. This experiment is otherwise identical to the samp_pc_ovfl experiment.

trace

When tracing experiments are done, selected functions are traced and produce a data record in the runtime experiment data file, if the function is executed.

The functions to be traced are defined by the pat_build -g, -u, -t, -T, or -w options specified when instrumenting the program. For more information about instrumenting programs for tracing experiments, see the pat_build(1) man page.

Note: Only true function calls can be traced. Function calls that are inlined by the compiler or that have local scope in a compilation unit cannot be traced.

Tracing experiments are also affected by the settings of other environment variables, all of which have names beginning with PAT_RT_TRACE_. These environment variables are described in the pat_build(1) man page.

PAT_RT_EXPFILE_APPEND

If nonzero, append the experiment data records to an existing experiment data file. If the experiment data file does not already exist, it is created.
If both **PAT_RT_EXPFILE_APPEND** and **PAT_RT_EXPFILE_REPLACE** are set, **PAT_RT_EXPFILE_APPEND** is ignored and the existing data file is replaced.

Default: 0

**PAT_RT_EXPFILE_DIR**

Identifies the path name of the directory in which to write the experiment file. If the name of the directory begins with the @ character, checks for ensuring that the directory resides on a record-locking file system, such as Lustre, are not performed.

**Note:** See **PAT_RT_EXPFILE_MAX** for more information about writing to a directory that resides on a file system that does not support record locking.

Default: the current execution directory

**PAT_RT_EXPFILE_FIFO**

If nonzero, the experiment data file is created as named FIFO pipe instead of a regular file. The instrumented program will block until the user executes another program that opens the pipe for reading. For more information, see the *mkfifo*(3) man page.

Default: 0

**PAT_RT_EXPFILE_FSLOCK**

Specifies the file record-locking attribute that overrides what CrayPat determines from evaluating the /etc/mtab file. This attribute indicates the type of file system locking supported. The valid values are:

- 0 or none: No file record-locking is supported.
- 1 or all: File record-locking is supported across all compute nodes and within the node itself.
- local: File record-locking is supported only within the node.

Default: unset

**PAT_RT_EXPFILE_MAX**

The maximum number of experiment data files created.

Default: 256
If the number of PEs used to execute the instrumented program is less than 256, a single data file per PE (up to 256 files) is created. If 256 or more PEs are used, the number of data files created is the square root of the number of PEs used for program execution, rounded up to the next integer value. If the value of PAT_RT_EXPFILE_MAX is $-1$ or greater than or equal to the number of PEs used for program execution, one data file per PE is created.

If PAT_RT_EXPFILE_MAX is set to 0, all PEs executing on a compute node write to the same data file. In this case the number of data files created depends on how the PEs are scheduled on the compute nodes and the directory need not reside on a file system that supports record locking.

**PAT_RT_EXPFILE_NAME**

Replaces the name portion of the experiment data file that was appended to the directory. The suffix, and other information, is appended to this name. If the value given to PAT_RT_EXPFILE_NAME ends with / or /+ any preceding name is interpreted as the name of a directory into which the experiment data files are written. If the name of the file begins with the @ character, the file is not removed if the instrumented program terminates during the initialization phase of CrayPat.

Default: the base name of the file

**PAT_RT_EXPFILE_PES**

Records data and writes the recorded data to its respective data file only for the specified PEs. If set to *, values from every PE are recorded.

Default: * (all PEs)

If not using the default, the PEs to be recorded are specified in a comma-delimited list, with each specification represented as one of the following:

- $n$ Value $n$.
- $n\sim m$ Values $n$ through $m$, inclusive.
- $n\%p$ Every $p^{th}$ value from 0 through $n$.
- $n\sim m\%p$ Every $p^{th}$ value from $n$ through $m$. 
For example, the following values are all valid specifications.

0, 4, 5, 10   Record PEs 0, 4, 5, and 10
15%4        Record PEs 0, 4, 8, and 12
4−31%8     Record PEs 4, 12, 20, and 28

**PAT_RT_EXPFILE_REPLACE**

If nonzero, replace an existing experiment data file with the new experiment data file. All data in the previous file is lost. If both **PAT_RT_EXPFILE_APPEND** and **PAT_RT_EXPFILE_REPLACE** are set, **PAT_RT_EXPFILE_APPEND** is ignored and the existing data file is replaced.

Default: 0

**PAT_RT_EXPFILE_SUFFIX**

The suffix component of the experiment data file name.

Default: .xf

**PAT_RT_EXPFILE_THREADS**

Record data for the specified thread only. If set to *, values from every thread are recorded.

Default: * (all threads)

If not using the default, the threads to be recorded are specified in a comma-delimited list, with each specification represented as one of the following:

- \( n \)  
  Value \( n \).

- \( n−m \)  
  Values \( n \) through \( m \), inclusive.

- \( n\%p \)  
  Every \( p^{th} \) value from 0 through \( n \).

- \( n−m\%p \)  
  Every \( p^{th} \) value from \( n \) through \( m \).

For example, the following values are all valid specifications.

0, 2     Record threads 0 and 2.
7%2    Record threads 0, 2, 4, and 6.
**PAT_RT_HEAP_BUFFER_SIZE**

Specifies the size in bytes of the runtime summary buffer used to collect dynamic heap information. This environment variable affects tracing experiments only.

Default: 2MB

**PAT_RT_INTERVAL**

Specifies the interval, in microseconds, at which the instrumented program is sampled.

To specify a random interval, use the following format:

\[ lower-bound, upper-bound[, \text{seed}] \]

After a sample is captured, the interval used for the next sampling interval is generated using \( \text{rand} \) and will be between \( lower-bound \) and \( upper-bound \). The initial seed (\text{seed}) for the sequence of random numbers is optional. See the \text{rand}(3) man page for more information.

This environment variable affects sampling experiments. It can also be used to control trace-enhanced sampling experiments, provided the program is instrumented for tracing but the \text{PAT_RT_EXPERIMENT} environment variable is used to specify a sampling-type experiment, and subject to the \text{PAT_RT_SAMPLING_MODE} environment variable setting.

Default: 10000 (microseconds)

**PAT_RT_INTERVAL_TIMER**

Specifies the type of POSIX interval timer used for sampling-by-time experiments. The following values are valid:

0 wall-clock (real) time

1 wall-clock (real) time guaranteed to be monotonic

This environment variable affects sampling experiments. It can also be used to control trace-enhanced sampling experiments, provided the program is instrumented for tracing but the \text{PAT_RT_EXPERIMENT} environment variable is used to specify a sampling-type experiment, and subject to the \text{PAT_RT_SAMPLING_MODE} environment variable setting. See the \text{timer_create}(2) man page for more information.

Default: 1
PAT_RT_MPI_MSG_BINS

Specifies the size boundaries of the histogram bins used to capture MPI messages sent between ranks. The specification is a comma-separated list of values. The maximum number of values indicating each bin size is 30. Zero and infinitely are implied.

**Note:** This environment variable affects data collection only when in runtime summary mode.

Default: 16, 256, 4kb, 64kb, 1mb, 16mb

PAT_RT_MPI_SYNC

Measure load imbalance in programs instrumented to trace MPI functions. If set to 1, this causes the trace wrapper for each collective subroutine to measure the time for a barrier call prior to entering the collective. This time is reported by `pat_report` in the function group `MPI_SYNC`, which is separate from the MPI function group.

If `PAT_RT_MPI_SYNC` is set, the time spent waiting at a barrier and synchronizing processes is reported under `MPI_SYNC`, while the time spent executing after the barrier is reported under `MPI`.

To disable measuring MPI barrier and sync times, set this environment variable to 0. This environment variable affects tracing experiments only.

Default: 1 (enabled)

PAT_RT_MPI_THREAD_REQUIRED

Specifies the MPI thread-level support for the instrumented program to use. This is a cardinal number that represents the MPI thread-level support. For more information, see the `MPI_Init_thread(3)` man page.

Default: As specified in the `MPI_Init_thread` function call in the original program

PAT_RT_PARALLEL_MAX

Specifies the maximum number of unique call site entries to collect for any OpenMP trace points generated by the CCE or PGI compilers when the OpenMP programming model is used. A call site is the text address at which the respective OpenMP trace point is called.

See the `pat_build(1)` man page for more information about compiler-generated trace points.

Default: 1024
PAT_RT_PERFCTR

Specifies the performance counter events to be monitored during the execution of a program instrumented for tracing experiments.

Counter events are specified in a comma-separated list. Event names and groups from all three components may be mixed as needed; the tools is able to parse the list and determine which event names or group numbers apply to which components. To list the names of the individual CPU, GPU, AMD Northbridge, power management, or network events on your system, use the `papi_avail(1)` and `papi_native_avail(1)` commands.

For lists of available network performance counters:

- On Gemini-based systems, either read the technical note *Using the Cray Gemini Hardware Counters*, read the files
  `$CRAYPAT_ROOT/share/Counters.papi_gemini` or
  `$CRAYPAT_ROOT/share/Counters.papi_gemini.xml`, or view the `counters→gemini` topics in `pat_help`.

- On Aries-based systems, either read the technical note *Using the Aries Hardware Counters*, read the files
  `$CRAYPAT_ROOT/share/Counters.papi_aries` or
  `$CRAYPAT_ROOT/share/Counters.papi_aries.xml`, or view the `counters→aries` topics in `pat_help`.

Depending on the counter selected, individual counter events are specified in one of three ways:

- use the performance counter event name, as given by `papi_avail` or `papi_native_avail`

- use the performance counter event name followed by the @ symbol and a value, to indicate a non-default overflow value used by the sampling-by-overflow experiments

- use the performance counter event name followed by the = sign and a value, to assign a value to a configuration event on an Aries network router

Alternatively, counter group numbers can be used in addition to or in place of individual event names, to specify one or more predefined performance counter groups. The valid counter CPU and GPU group numbers are listed in the `hwpc(5)` and `accpc(5)` man pages respectively. Predefined network counter groups have names instead, and are listed in the `nwpc(5)` man page.
In addition, this environment variable supports the use of keywords. The keywords currently recognized are:

- `domain:u` — specify that hardware counters are active in the user's domain
- `domain:k` — specify that hardware counters are active in the kernel (OS) domain
- `domain:x` — specify that hardware counters are active in the exception domain
- `mpx` — enable multiplexing for CPU events

Default: unset

**PAT_RT_PERFCTR_FILE**

Specifies, in a comma-separated list, the names of one or more files that contain performance counter specifications. Within the files, lines beginning with the # character are interpreted as comments and ignored. See PAT_RT_PERFCTR for a description of an event specification.

Default: unset

**PAT_RT_PERFCTR_FILE_GROUP**

Specifies, in a comma-separated list, the names of one or more files that contain performance counter group definitions. A group definition consists of at least one valid performance counter event. Use the `papi_avail` and `papi_native_avail` commands to determine the names of valid events.

The format of the file is: `group-name=event1,...`

The definition of the group is terminated with a `<newline>` character. There may be multiple unique group names defined in a single file. Lines that do not match this syntax are ignored.

If the first file name in the list is the character 0 (zero), the default counter groups are not loaded and therefore are not available for selection using PAT_RT_PERFCTR.

The file containing the group definitions for the default groups is in `$CRAYPAT_ROOT/share/`.

Default: unset
PAT_RT_RECORD

Specifies the initial data collection and recording state for the instrumented program. If set to zero, no performance data is collected or recorded when the program starts execution. Use the PAT_record API call to turn on data collection and recording; see the pat_build(1) man page for more information.

Default: unset

PAT_RT_REGION_CALLSTACK

Specifies the depth of the stack for which the CrayPat API functions PAT_region_begin and PAT_region_end are maintained. In other words, it is the maximum number of consecutive PAT_region_begin references that can be made without an intervening PAT_region_end. Setting this environment variable to zero (0) disables data collection for all regions. This environment variable affects tracing experiments only.

Default: 128

PAT_RT_REGION_MAX

Specifies the largest numerical ID that may be used as an argument to the CrayPat API functions PAT_region_begin and PAT_region_end. Values greater than this cause the API function to be ignored. Setting this environment variable to zero (0) disables data collection for all regions. This environment variable affects tracing experiments only.

Default: 100

PAT_RT_REPORT_CLEANUP

If the report directive is set to y in pat_build when the program is instrumented, a textual report is written to stdout when the instrumented program successfully completes execution. This environment variable specifies how the temporary files used for report generation are removed after the report is produced. The valid values are skip, fail, and force, where skip does not remove any files, fail removes files only if there is an error in report generation, and force always removes files.

Default: fail
**PAT_RT_REPORT_CMD**

This environment variable supports two or more comma-separated arguments, `report-command` and `report-options`, which can be used to specify the pathname of the executable file that produces the text report and then a comma-separated list of one or more report options to be passed to `pat_report`.

If only `report-command` is set, a default text report is produced when the program terminates successfully. If `report-options` are also included, you can control the content and format of the resulting report. The valid `report-options` options are listed in the `pat_report(1)` man page.

Defaults:

- `report-command` — `$CRAYPAT_ROOT/bin/pat_report`
- `report-options` — none

**PAT_RT_REPORT_METHOD**

If the `report` directive is set to `y` in `pat_build` when the program is instrumented, a textual report is written to stdout when the instrumented program successfully completes execution. This environment variable defines the mechanism used to create the text report. Valid values are `pe0` and `team`. The `pe0` argument uses only PE zero to control all aspects of report generation, while the `team` argument uses all PEs to share control of all aspects of report generation. To disable report generation, set this environment variable to 0.

**Note:** Implementation of the `team` argument is deferred.

Default: `pe0`
PAT_RT_SAMPLING_MODE

Specifies the mode in which trace-enhanced sampling operates. Trace-enhanced sampling allows a sampling experiment to be executed on a program instrumented for tracing. It affects both user-defined functions and predefined function groups. The value may be one of the following.

0  Ignore trace-enhanced sampling. The normal tracing experiment is performed.
1  Enable raw sampling. Any traced entry points present in the instrumented program are ignored.
3  Enable bubble sampling. Traced entry points and any functions they call return a sample PC address mapped to the traced entry point.

When set to a nonzero value, all sampling experiments and parameters that control sampling apply to the executing instrumented program. Tracing records are not produced.

Default: 0

PAT_RT_SAMPLING_SIGNAL

Specifies the signal that is issued when a POSIX interval timer expires or a hardware performance counter overflows.

This environment variable affects sampling experiments. It can also be used to control trace-enhanced sampling experiments, provided the program is instrumented for tracing but the PAT_RT_EXPERIMENT environment variable is used to specify a sampling-type experiment, and subject to the PAT_RT_SAMPLING_MODE environment variable setting.

This environment variable accepts the names of signals as given in the signal(7) man page; for example, SIGALRM, SIGPROF, etc. The signal as specified as a cardinal number is also accepted. Note that a given signal may be used by other components or features of the instrumented program, and some signals may interfere with CrayPat initialization or runtime data collection.

Default: 27 (SIGPROF)
PAT_RT_SAMPLING_MASK

Specifies a bitmask that is AND'd with the PC address acquired during a sampling experiment. This can reduce the number of unique addresses collected. The default value is 0xffffffff and is specified in hexadecimal notation.

PAT_RT_SETUP_SIGNAL_HANDLERS

If zero, the CrayPat runtime library does not catch signals that the program receives; this results in an incomplete experiment file but a more accurate traceback for an aborted program with a core dump.

Default: 1

PAT_RT_SUMMARY

If set to a nonzero value, runtime summarization is enabled and the data collected is aggregated. This greatly reduces the size of the resulting experiment data files but at the cost of fine-grain detail, as formal parameter values, function return values, and call stack information are not recorded.

If set to 0, runtime summarization is disabled and performance data is captured in detail.

Disabling runtime summarization can be valuable, particularly if you plan to use Cray Apprentice2 to study your data. However, be advised that setting this environment variable to 0 can produce enormous experiment data files, unless you also use the CrayPat API to limit data collection to a specified region of your program.

Default: 1 (enabled)

PAT_RT_THREAD_ALLOW

Specifies how created threads are monitored and recorded. If set to a nonzero value, every thread created after the main entry point has executed is monitored and its data recorded. Set to zero to ignore all data collection for created threads.

Default: 1 (enabled)
PAT_RT_THREAD_CANCEL_NTRIES

Specifies the number of attempts the main thread makes in waiting for all created threads to terminate. An attempt is made every 0.25 seconds. Once all attempts have been completed by the main thread, the rest of the shutdown procedures can complete.

**Note:** Once the shutdown procedures begin, any thread that has not terminated is forced to exit, possibly causing the thread's collected data not to be recorded in the data file.

Default: 120 (30 seconds)

PAT_RT_THREAD_MAX

Specifies the maximum number of threads that can be created and for which data is recorded. See PAT_RT_EXPFILE_THREADS to manage the recording of data for individual threads.

Default: 1,000,000

PAT_RT_TRACE_API

If 0, suppress the events and any data records produced by all embedded CrayPat API functions in the instrumented program. For more information about the CrayPat API, see the pat_build(1) man page.

Default: 1 (enabled)

PAT_RT_TRACE_DEPTH

Specifies the maximum depth of the runtime call stack for traced functions during runtime summarization.

Default: 512

PAT_RT_TRACE_FUNCTION_ARGS

Specifies the maximum number of function argument values recorded each time a function is called during a tracing experiment. This environment variable applies to tracing experiments only and is ignored in trace summary mode.

Default: all argument values to a function are recorded in full trace mode
**PAT_RT_TRACE_FUNCTION_DISPLAY**

If set to a nonzero value (enabled), write the function names which have been instrumented in the program to stdout. This environment variable affects tracing experiments only.

Default: 0 (disabled)

**PAT_RT_TRACE_FUNCTION_MAX**

The maximum number of traces generated for all instrumented functions for a single thread. This environment variable affects tracing experiments only.

Default: the maximum number of traces is unlimited

**PAT_RT_TRACE_FUNCTION_NAME**

Specify by name the instrumented functions to trace. This environment variable replaces PAT_RT_FUNCTION_LIMITS. The value is a comma-separated list of one of two forms:

```
function-name1,
...,
function-namen

or

function-name,
function-name: last
```

In the first form tracing records are produced every time the instrumented function `function-name` is executed. In the second form tracing records are produced only for the instrumented function `function-name` until `function-name` is executed `last` number of times.

If the function name is `*`, any value specified applies to all instrumented functions. For example

`*:0`

prevents all instrumented functions from recording trace data, whereas

`*:0,function-name`

specifies that only the instrumented function `function-name` will record trace data. This environment variable affects tracing experiments only.

Default: unset
**PAT_RT_TRACE_FUNCTION_SIZE**

Specify the size in bytes of the instrumented function to trace in a program instrumented for tracing. The size is given as min, max, where min is the lower limit and max is the upper limit, specified in bytes. A trace record is produced only when the size of the instrumented function lies between min and min, max. This environment variable affects tracing experiments only.

Default: unset

**PAT_RT_TRACE_HEAP**

If set to 0, disable the collection of dynamic heap information. This environment variable affects tracing experiments only.

Default: 1 (enabled), if malloc is present

**PAT_RT_TRACE_HOOKS**

Enable/disable instrumentation inserted as a result of tracing options specified when compiling the program. (See pat_build(1).) The syntax is a comma-separated list of compiler instrumentation types and toggles in the form name:a,name:a..., where name represents the nature of the compiler instrumentation and a is either zero to disable the specified event or nonzero to enable it. If no name is specified and PAT_RT_TRACE_HOOKS is set to zero, all compiler-instrumented tracing is disabled.

**Note:** PAT_RT_TRACE_HOOKS interacts with PAT_RT_SUMMARY. For more information, see Default, below.

The valid values for name are:

- **acc** GPU accelerator events
- **chapel** Chapel events
- **func** function entry and return events
- **loops** loop timing events
- **omp** OpenMP events
Default: 1 (collect data for all compiler-inserted trace points) if `PAT_RT_SUMMARY` is unset or set to a nonzero value (that is, if run time summarization is enabled); `acc:1,omp:1` (collect data for GPU accelerator events and OpenMP events but ignore all other compiler-inserted trace points) if `PAT_RT_SUMMARY` is set to 0 (that is, if run time summarization is disabled).

**PAT_RT_TRACE_OVERHEAD**

Specify the number of times the functions used to calculate the calling overhead are called upon runtime initialization and termination. To suppress overhead calculations, set this to 0. The larger the value, the more accurate the overhead calculation.

Default: 100

**PAT_RT_TRACE_THRESHOLD_PCT**

Specify a threshold to enforce when executing in full trace mode. The format is `ncalls,pct` where `pct` is between 1 and 100. If a function's total time relative to its executing thread's total time falls below the percentage `pct`, trace records for the function are no longer produced. The function must be called at least `ncalls` time(s) in order to activate the threshold.

For example, if `PAT_RT_TRACE_THRESHOLD_PCT` is set to 1000,15, and a function's total time relative to the executing thread's time falls below 15 percent after being called at least 1,000 times, trace records for the function are no longer written to the experiment data file.

This environment variable affects tracing experiments only.

Default: unset

**PAT_RT_TRACE_THRESHOLD_TIME**

Specify a threshold to enforce when executing in full trace mode. The format is `ncalls,microsecs`. If a function's average time per call falls below the time specified by `microsecs`, trace records for the function are no longer produced. The function must be called at least `ncalls` time(s) in order to activate the threshold.

For example, if `PAT_RT_TRACE_THRESHOLD_TIME` is set to 2500,500, and a function's average time per call falls below 500 microseconds after being called at least 2,500 times, trace records for the function are no longer written to the experiment data file.

This environment variable affects tracing experiments only.
Default: unset

**PAT_RT_VALIDATE_SYSCALLS**

If set to 0, prevent the instrumented program from executing selected function calls that can interfere with the instrumented program's runtime data collection. The selected function calls include `profil`, `setitimer`, `signal`, and `sprofil`.

**Note:** Using this option to block function calls may cause unexpected behavior and interfere with runtime data collection.

Default: 1 (function calls enabled)

**PAT_RT_VERBOSE**

If set, specify the PEs from which to accept and record info-level messages. If set to *, messages from every PE are accepted.

Default: unset

Alternatively, the PEs to be recorded can be specified in a comma-delimited list, with each specification represented as one of the following:

- $n$ PE $n$.
- $n$–$m$ PEs $n$ through $m$, inclusive.
- $n$%$p$ Every $p^{th}$ PE from 0 through $n$.
- $n$–$m$%$p$ Every $p^{th}$ PE from $n$ through $m$.

**PAT_RT_WRITE_BUFFER_SIZE**

Specify the size, in bytes, of a buffer that collects measurement data for a single thread.

Default: 8MB
A.3 pat_report Environment Variables

The pat_report environment variables affect the way in which data is handled during report generation.

PAT_REPORT_IGNORE_VERSION
PAT_REPORT_IGNORE_CHECKSUM

If set, turns off checking that the version of CrayPat being used to generate the report is the same version, or has the same library checksum, as the version that was used to build the instrumented program.

PAT_REPORT_OPTIONS

If the -z option is specified on the pat_report command line, this environment variable is ignored.

If the -z option is not specified, then, if this environment variable is set before pat_report is invoked, the options in this environment variable are evaluated before any other options on the command line.

If this environment variable is not set when pat_report is invoked, but was set when the instrumented program was run, then the value of this variable as recorded in the experiment data file is used.

PAT_REPORT_PRUNE_NAME

Prune (remove) functions by name from a report. If not set or set to an empty string, no pruning is done. Set this variable to a comma-delimited list (__pat__, __wrap__, etc.) to supersede the default list, or begin this list with a comma (,) to append this list to the default list. A name matches if it has a list item as a prefix.

PAT_REPORT_PRUNE_SRC

If not set, the behavior is the same as if set to '/lib'.

If set to the empty string, all callers are shown.
If set to a non-empty string or to a comma-delimited list of strings, a sequence of callers with source paths containing a string from the list are pruned to leave only the top caller.

**PAT_REPORT_PRUNE_NON_USER**

If set to 0 (zero), disables the default behavior of pruning based on ownership (by user invoking `pat_report`) of source files containing the definition of a function.

**PAT_REPORT_VERBOSE**

If set, produces more feedback about the parsing of the `.xf` file and includes in the report the values of all environment variables that were set at the time of program execution.
Cray XK and Cray XC30 systems include GPU accelerators. To take advantage of these accelerators, programmers must modify their code, either by inserting Cray CCE OpenACC directives, PGI accelerator directives, or CUDA driver API code.

For the most part, the Cray Performance Analysis Tools behave the same on Cray XK and Cray XC30 systems and with accelerated code as they do on Cray XE systems and with conventional code, with the following caveats, exceptions, and differences.

B.1 Module Load Order

In order for the Cray Performance Analysis Tools to function correctly on Cray XK and Cray XC30 systems, module load order is critical. Always load the accelerator target module *before* loading the performance tools module. The following example shows a valid module loading sequence for compiling and instrumenting code to run on a Cray XK systems equipped with AMD Interlagos CPUs and NVIDIA K20 Tesla GPUs.

```
> module load PrgEnv-cray
> module load craype-interlagos (optional)
> module load craype-accel-nvidia20
> module load perftools
```

On actual Cray systems, the correct `craype` module for the type of CPU installed on the system compute nodes is typically loaded by default; therefore it is not necessary for the user to load the module. On esLogin systems and standalone Linux systems being used as cross-compiler code development workstations, it may be necessary to load the appropriate CPU target (`craype`) module, depending on the local configuration. Always verify that you have the correct CPU target module loaded for the Cray system on which you will be executing the resulting code, as the choice of CPU target module can have very significant impact on the behavior and execution speed of the resulting compiled code.
B.2 pat_build Differences

In general, pat_build behaves the same with code containing compiler accelerator directives or CUDA driver API code as it does with conventional code. There are no pat_build options unique to Cray XK or Cray XC30 systems.

Note: Accelerated applications cannot be compiled using the CCE `-h profile_generate` option, therefore accelerator performance statistics and loop profile information cannot be collected simultaneously.

B.3 Run Time Environment Differences

The CrayPat run time environment supports three environment variables that apply to Cray XK and Cray XC30 systems only. These are:

PAT_RT_ACC_ACTIVITY_BUFFER_SIZE

Specifies the size in bytes of the buffer used to collect records for the accelerator time line view in Cray Apprentice2. Size is not case-sensitive and can be specified in kilobytes (KB), megabytes (MB), or gigabytes (GB).

Default: 1MB

PAT_RT_ACC_RECORD

Overrides the programming model for which accelerator performance data is collected. The valid values are:

- off: Disables collection of accelerator performance data.
- cce: Collect performance data for applications compiled with CCE and using OpenACC directives.
- cuda: Collect performance data for CUDA applications.
- pgi: Collect performance data for applications using PGI accelerator directives.

Default: unset

PAT_RT_ACC_FORCE_SYNC

Forces accelerator synchronization in order to enable collection of accelerator time for asynchronous events.

Default: not enabled
B.4 pat_report Differences

Assuming data was collected for accelerator regions, pat_report automatically produces additional tables showing performance statistics for the accelerated regions. In addition, pat_report now includes six new predefined reports that apply to Cray XK and Cray XC30 systems only. These are:

- accelerator
  - Show calltree of GPU accelerator performance data sorted by host time.
- accpc
  - Show accelerator performance counters.
- acc_fu
  - Show accelerator performance data sorted by host time.
- acc_time_fu
  - Show accelerator performance data sorted by accelerator time.
- acc_time
  - Show calltree of accelerator performance data sorted by accelerator time.
- acc_show_by_ct
  - (Deferred implementation) Show accelerator performance data sorted alphabetically.

B.5 Cray XC30 CPU Hardware Counter Differences

Because Cray XC30 systems use Intel processors, there are significant differences in the hardware counters available for use. For more information, see the hwpc(5) man page.

On Cray systems with Intel Sandybridge processors, the values reported for floating point operations may be significantly larger than the number of operations actually specified in the program. There are two reasons for this. First, operations must be calculated from instruction counts that include speculatively issued instructions. Second, for the general case, more counts are required than can be supported by the physical hardware counters, and so PAPI multiplexing is used for the CrayPat default event set. If it is known that, for example, only single precision operations are of interest, then a smaller set of events can be used, which can be counted without multiplexing.

Note the following details:

1. Floating point operations cannot be counted directly, but the various types of floating point instructions can be counted, and so an operation count can be calculated with a weighted sum, where each summand is an instruction count times the number of operations resulting from one instruction of that type.
2. For a weighted sum for all types of floating point operations, it would suffice to get combined counts for all instructions that produce the same number of operations. This would reduce the number of events that must be counted.

3. The reduction in the number of events described in point 2 is limited by the facts that subevents of `FP_COMP_OPS_EXE` and `SIMD_FP_256` cannot be combined, and that at least one combined event, `FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE:SSE_SCALAR_DOUBLE`, does not produce correct results.

4. With hyper-threading enabled, the number of physical counters available for FP events is 4, and this is not enough to accommodate the events required for the weighted sum. So either multiplexing must be used or multiple runs must be made to count subsets of these events. In order to give at least approximate values from a single run, the CrayPat default event set uses multiplexing.

These details were discovered independently by CrayPat developers experimenting with simple computational kernels, but have been reported by other groups as well. For more information, see the PAPI website at http://icl.cs.utk.edu/papi/.

### B.6 Cray XC30 CPU Network Counter Differences

Because Cray XC30 systems use the Aries interconnect, there are significant differences in the network counters available for use. For more information, see the `nwpc(5)` man page. For more detailed information about the individual counters that make up the groups, see `$CRAYPAT_ROOT/share/CounterGroups.aries`.

For in-depth information about the Aries Performance Counters, see the technical note *Using the Aries Hardware Counters (S–0045)*, available on the Cray website.